Journal of Universal Computer Science, vol. 15, no. 3 (2009), 620-647
submitted: 25/8/08, accepted: 30/1/09, appeared: 1/2/09

Supporting Composition of Structural Aspects in an AOP Kernel

Éric Tanter[1] and Johan Fabry[2]
(PLEIAD Laboratory
Computer Science Department (DCC), University of Chile
Santiago, Chile
etanter@dcc.uchile.cl, jfabry@dcc.uchile.cl
http://pleiad.cl/)

Abstract: Structural aspects modify the structure of a program, for instance by adding fields and methods to existing classes. Like behavioral aspects, which operate on execution events, structural aspects may interact and raise conflicts. Current aspect systems however do not thoroughly handle this issue. This paper discusses how complete support for structural aspect composition can be integrated in an AOP kernel, that is, a generic transformation framework on top of which aspect languages are defined. An iterative composition process is proposed that involves the programmer in a cycle of automatic detection of interactions and explicit, declarative resolution of these interactions. Beyond a general analysis of the issue of composition of structural aspects and an associated composition process, this work reports on the concrete extension of the Reflex AOP kernel to fully support the requirements drawn from our analysis. Based on a structural model supporting per-aspect subjective views, and using the power of an embedded logic engine, the result is a versatile aspect system supporting automatic detection of various kinds of structural aspect interactions, extensible reporting tools, and declarative mechanisms for the resolution of interactions between structural aspects.

Key Words: Aspect-Oriented Programming, AOP kernel, structural aspects, aspect composition, Reflex

Category: D.1.5, D.2.3, D.3.3, D.3.4

1 Introduction

Aspect-Oriented Programming (AOP) provides means for the proper modularization of crosscutting concerns [Elrad et al. 2001]. The fact that many aspects can be applied to the same program raises the aspect *composition* issue [Bussard et al. 2000], which is increasingly attracting the attention of the research community, as the use of AOP increases and scaling issues arise.

Most AOP approaches focus on *behavioral* aspects following the pointcut-advice model of AspectJ [Kiczales et al. 2001], not much considering *structural* aspects, as exemplified by inter-type declarations (*aka.* introductions) in AspectJ. A structural aspect is one that, as part of its action, modifies the structure

[1] Éric Tanter is partially funded by FONDECYT project 11060493 and 1090083
[2] Johan Fabry is partially funded by FONDECYT project 1090083

of program elements. Usually, structural aspects in current aspect languages are able to *add* members or interfaces to classes. Informal evidence however strongly suggests that a large part of the use of AspectJ in real-life applications consists of such structural aspects.

The popularity of the use of structural aspects is in stark contrast to the support given for their composition. Indeed, most work on composition of aspects simply ignores issues related to structural aspect composition [Klaeren et al. 2000, Brichau et al. 2002, Douence et al. 2002, Nagy et al. 2004, Douence et al. 2004]. Only recently some proposals emerged that deal with structural aspect composition [Havinga et al. 2006, Tanter 2006a, Lopez-Herrejon et al. 2006, Harrison et al. 2006]. Structural interactions can arise in various ways. First, because of aspects modifying base code in conflicting ways, yielding invalid code; for instance by adding a method to a class that already contains a method with the same signature. Also, because aspects typically rely on structural properties of a program (possibly augmented with dynamic properties) in order to determine if they apply, the fact that some aspects may alter this structure can result in inconsistencies and surprises due to (hidden) dependencies.

For instance, if the KALA domain-specific aspect language for advanced transaction management [Fabry et al. 2008] is used to make all methods of a class transactional, while another aspect adds a method to the same class, what should happen with this added method with respect to transactionality? Based on concrete experiments with multi-language AOP [Tanter and Noyé 2005, Fabry et al. 2009], we have started to analyze and address the issue of aspect composition [Tanter 2006a]. This work is an extension of previous work that focuses on structural aspect composition.

In [Tanter and Noyé 2005] we have argued for the use of a versatile AOP kernel to allow for faster construction of AOP languages, called Reflex. Reflex is a kernel for multi-language AOP in Java, that is, an AOP system whose aim is to facilitate the definition and integration of different aspect languages, including domain-specific ones, to modularize the different concerns of a software system. Using Reflex as an underlying kernel, an aspect language implementation can rely on its implementation of interaction. In previous published work [Tanter and Noyé 2005, Tanter 2006a, Tanter et al. 2008], Reflex however did not provide complete support for the composition of structural aspects. This paper discusses the work performed to provide full support for composition of structural aspects in Reflex.

To provide such support, we studied the case of interactions between structural aspects in detail. The goal was to achieve a general model for these interactions, and allow resolution through explicit specification, as shown to be required by Douence *et al.* [Douence et al. 2002]. We shall discuss in this paper that this raises several issues which are not found in the case of behavioral aspects.

As a result of this work, Reflex now contains interaction detection and resolution features for structural aspects. These features provide a uniform interface to the implementer of an aspect language. The only work that needs to be done by the aspect language implementer to fully enable this feature, is to reify the explicit specification of resolution in the aspect language (with simplifications, if deemed necessary).

The contributions of this work are:

– An analysis of interactions between structural aspects, which results in the identification of *(a)* three kinds of interactions depending on the parties involved (base-action, action-action, action-cut), *(b)* four possible interaction resolution mechanisms (skipping actions, combining elements, visibility of changes, and order of application), as well as *(c)* a clear distinction of three dimensions of interactions (conflicting, resolved, effective).

– The proposition of a composition process that explicitly considers the levels of involvement of the programmer and the iterative nature of the detection and resolution of interactions.

– A full implementation of the proposed process in Reflex, supporting *(a)* a uniform representation of all the identified interaction kinds, allowing extensible reporting tools to be developed, *(b)* the automatic detection of interactions, including action-cut interactions, based on a logic engine integrated into Reflex, *(c)* the different declarative resolution mechanisms previously highlighted.

– The illustration of how advanced language mechanisms, such as subjective views over the program structure, and collaboration between an object-oriented model and a logic engine, can be leveraged to address some of the challenges raised by scaling up aspect-oriented programming.

The structure of the paper is as follows: Section 2 reports, in a general setting, on structural aspect interactions, their kinds, properties, and possible resolution mechanisms. Section 3 proposes, also in a general setting, an iterative process to support composition of structural aspects. Section 4 presents our general approach to structural aspect composition in the case of Reflex, providing background information on how structural aspects are supported by this platform. Then, we present our proposal in more details: Section 5 focuses on the automatic detection of structural aspect interactions, Section 6 on the reporting of interactions to the programmer, and Section 7 on the resolution mechanisms, both from the point of view of the programmer and from the point of view of how they are implemented in the Reflex kernel. Section 8 discusses related and future work, and Section 9 concludes.

2 Interactions of Structural Aspects

In this paper, a *structural element* denotes any piece of structure in an object-oriented program, *i.e.* a class, interface, annotation, field, method, constructor, or expression. A *structural container* is a structural element containing other structural elements; for instance, a class is a structural container of members, and a member is a structural container of its annotations and body expressions.

A *structural interaction* is an interaction involving the structural modification performed by an aspect. Behavioral interactions, *i.e.* interactions involving behavioral modifications by aspects, refer to the problem of *shared join points*, *e.g.* two aspects that affect the same method execution. In other words, behavioral interactions dealt with in the literature are typically *cut-cut interactions*: the cuts of two aspects overlap. For brevity in this paper we call structural modifications of an aspect *structural aspects* and behavioral modifications of an aspect *behavioral aspects*[3].

Cut-cut interactions of structural aspects are not relevant as such because two structural aspects can apply *orthogonally* to the same class (*e.g.* by adding two completely unrelated methods). The interest is rather in dealing with interactions involving the *action* of at least one structural aspect. Such an action can either interact with the base code, or with the action or cut of other aspects.

In this section, we study interactions of structural aspects in more detail. We ignore behavioral interactions, as these will not be covered by our model. We first clarify what we mean by structural aspects, highlighting the range of our analysis and proposal. We then propose a classification of structural interactions (Section 2.2). We discuss the detection and resolution mechanisms that one would expect from a comprehensive system fully supporting structural aspect composition (Section 2.3). Finally, in Section 2.4 we come back to the terminology in light of the previous section.

2.1 Anatomy of Structural Aspects

Considering structural aspects, we distinguish two levels of aspects. A *primitive* aspect is a single pair consisting of a *cut* and an *action*. The cut of an aspect is the (usually intensional) selection of points of interest, either static or dynamic. The action is the specification of the effect of the aspect on its cut. In this

[3] It has to be kept in mind that a single aspect, as the modular implementation of a crosscutting concern, can be both behavioral and structural. For instance, a single AspectJ aspect can define inter-type declarations (structural) as well as pointcuts and advices (behavioral).

view, a primitive aspect is said to be behavioral if its action affects the *behavior* of the application, and *structural* if its action modifies the program structure. These cut-action pairs are said to be primitive because most aspects are indeed composed of several such pairs. For instance an AspectJ aspect can perform a number of inter-type declarations as well as define a number of pointcuts and advices. Such a *composite* aspect can be viewed as grouping several structural and behavioral primitive aspects. This view is useful because, as said above, the actual kinds of interactions and ways to handle composition differ greatly enough between structural and behavioral aspects to deserve separate treatment.

In this work we limit our analysis to structural aspects whose cut relies on structural introspection (*i.e.* lexical information): the cut of a structural aspect is a condition over the properties of the structural elements that make up the program. Also, the cut of a structural aspect is considered to be as expressive as needed: in other words, the cut is possibly algorithmic, defined in a Turing-complete language, and has a full power of introspection, down to expressions, as proposed in [Gybels and Brichau 2003]. This supports what is also known as *expressive pointcuts* in behavioral aspects [Masuhara and Aotani 2006]. For structural aspects in AspectJ, the cut is defined by a *type pattern*, which is insufficient for expressive cuts such as "matching classes that have at least one method that does at least one message send matching a given pattern". Expressive cut for structural aspects is also provided by Josh [Chiba and Nakagawa 2004].

The actions we consider are the *addition* of structural elements to a structural container, *e.g.* adding a new class, a new method to a class, or a new annotation to a field. This corresponds to the sum introduction operator in the algebra presented in [Lopez-Herrejon et al. 2006], and concretely implemented in a language like AspectJ. Note that we consider neither addition of expressions nor modifications like renaming and removing. The precise analysis of the consequences of these features on composition support is left as future work. Even considering the above restrictions, the present work covers current proposals of structural aspects like inter-type declarations of AspectJ and more, due to the fact that structural cuts and actions are operationally defined in full Java.

2.2 Kinds of Structural Interactions

Recall that our interest is in dealing with interactions involving the action of at least one structural aspect, which can either interact with the base code, or with the action or cut of other aspects. The key property of structural aspects here is that their *action domain* is *bounded*: structural actions here are only modifications of the program structure. This is an action domain that is restricted enough to be analyzed in order to fully address interactions. This analysis yields the following three kinds of structural interactions:

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.

Figure 1: Base-action interactions.

Base-action interactions. This kind of interaction refers to clashes between structural elements added by aspects on the one hand, and structural elements of the original base code on the other hand (Fig. 1). Examples of such interactions include an aspect adding a class that has the same name as an existing class, or adding a method to a class that already has one with the same signature.

Action-action interactions. This kind of interaction refers to clashes between structural elements added by two aspects (Fig. 2). Such an interaction occurs for instance if two aspects add methods with the same signature to the same base class, or add the same annotation to the same structural element.

Action-cut interactions. This kind of interaction refers to potential dependencies between the (intensionally-defined) cut of a structural aspect (*i.e.* the set of structural elements it affects) and structural elements newly introduced by another aspect (Fig. 3). The question being raised is whether the introduced element should possibly be part of the cut of other aspects. Examples of such interactions include an aspect adding a class to a given package, while another aspect adds a method to all classes of that package —*should the introduced class get the new method?*—; or an aspect adding an annotation to all fields of a class, and an aspect adding a field to that class —*should the introduced field be annotated?*—.

These are the three kinds of aspect interactions that can occur at the structural level, addressed in this work. It has to be made clear that more general semantic conflicts between aspects, for instance when an aspect introduces a method whose execution contradicts some invariants enforced by another aspect, are out of the scope of this work. This general semantic conflict issue is as vast as addressing general correctness issues of programs.

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.

Figure 2: Action-action interactions.

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.

Figure 3: Action-cut interactions.

2.3 Detection and Resolution Mechanisms

To allow us to build a general model for the composition of structural aspects, we need to consider the different possible detection and resolution mechanisms for each of the interaction kinds discussed above.

The two first kinds of interaction we mentioned (Fig. 1 and Fig. 2) typically result in compilation errors, as the underlying processor (compiler/interpreter) rejects the addition of an already-existing structural element. So their detection is in a way ensured by traditional technology. There are therefore two major alternatives: *(1)* ensuring that the interaction does not occur, either by manually modifying the definition of (one of) the aspect(s), or by declaring that (one of) the aspect action(s) should be skipped; *(2)* specifying an actual *combination* of the structural elements in conflict. Manual modification of the aspect(s) to avoid the interaction does not deserve any special mechanism from the aspect system, so we do not discuss it further. We are therefore left with the two following desirable resolution mechanisms:

Skipping aspect actions. This mechanism consists in specifying that the detected interaction should not happen. This can possibly be declared either by stating that an aspect does not apply to the class causing the problem, or that every conflict provoked by the aspect should be skipped (*i.e.* by simply not adding the method that the aspect was supposed to add). Another point of view is to declare some mutual exclusion between two aspects, by stating that if two aspects apply on the same class, one of them has to be skipped. A variant is to generate an error in such a case, for instance if two aspects are known to be intrinsically incompatible and hence any interaction between them is to be considered a programming error.

Combining structural elements. When the addition of a structural element conflicts with the current state of the program, a possibility is to *combine* the element to add with the existing one (which may have been added by an aspect or not). By combining, we mean a mechanism similar to the composition operator provided in traits [Schärli et al. 2003], where two conflicting methods can be aliased and used in a third combination method. When taken to the level of classes, this mechanism resembles the composition mechanisms offered in systems like Hyper/J [Ossher and Tarr 2001, Harrison et al. 2006].

Action-cut interactions (Fig. 3) are more subtle, because they generally do not result in compilation errors. Still, they can have important *semantic* impact. For instance, in the second example of Fig. 3, it is important that the programmer is informed that the field added by `A2` to `C` *may or may not* be annotated by `A1`. The aspect system has absolutely no means to automatically infer the desired semantics, as it all depends on the particular application and setting. This implies that it is crucial for the programmer that the aspect system *detects* them. We identify two dimensions to the possible resolution mechanism:

Visibility of changes. The first dimension concerns the visibility of structural changes made by an aspect to the cut of other aspects. Some changes may necessarily be hidden, others visible, while some changes may potentially be visible to only *some* aspects and not others.

Order of application. The second dimension relates to the order in which structural aspects are applied, that is, the order in which changes to a class definition are effectively carried out. This may affect the correct compilation of some changes, for instance if a method added by an aspect has a reference to another method that is added by another aspect, the referred method must be added before the other one.

These two dimensions are clearly not orthogonal. If it is ensured that all structural changes made by aspects are *invisible* to other aspects, then there cannot be any action-cut interaction. However, if *some* structural changes made by an aspect `A` can be seen by all or some other aspect(s), then necessarily aspect `A` has to be applied *before* these other aspects determine their cut. This discussion leads us to the need for a clarification of the terminology associated to interactions.

2.4 Terminology: Interactions and Conflicts

To clarify the different cases of interactions that can be faced, we introduce three independent dimensions. The first refers to the fact that an interaction can or cannot be an actual *conflict*:

Definition 1 Conflicting interaction. An interaction is *conflicting* (*a.k.a.* a conflict) if and only if it results in undesired semantics from the point of view of either the program processor or the programmer. Otherwise, it is said to be *non-conflicting*.

This dimension is important because it actually highlights that all interactions are not necessarily "problems" as such. The second dimension relates to the explicit specification of a resolution by the programmer:

Definition 2 Resolved interaction. An interaction is *resolved* if and only if the program code includes an *explicit specification* of the desired resolution. Otherwise, it is said to be *unresolved*.

Finally, as discussed in the previous section, action-cut interactions are subtle because they do not necessarily occur: contrarily to base-action and action-action interactions, their *effectiveness* depends on both visibility of changes and ordering of application:

Definition 3 Effective interaction. An interaction is *effective* if it can occur; otherwise, it is said to be *non-effective*. Base-action and action-action interactions are *always* effective. An action-cut interaction between the action of an aspect A and the cut of an aspect B is effective if and only if (1) the action of A is evaluated *before* the cut of B, and (2) the structural changes made by the action of A are *visible* to the cut of B.

Discriminating between these dimensions is important when it comes to considering the actual *composition process* in which a programmer has to engage.

3 An Iterative Composition Process

Section 2 has proposed an analysis of the issue of structural aspect interactions, from the point of view of the nature of the interactions and what can be done with respect to their detection and resolution. This section now approaches the problem from a higher-level point of view, and proposes a *composition process*.

This process clarifies the role and interactions between the programmer on the one hand, and the machinery for detection, resolution and actual weaving on the other hand. Our concrete implementation of this process, in the Reflex AOP kernel, is presented from Section 4 onward.

3.1 General Approach to Composition

Our approach to composition follows that proposed by Douence *et al.* in the context of behavioral aspects [Douence et al. 2002]: it relies on *automatic detection* of aspect interactions, *explicit resolution* of the interactions, and then composition by the aspect system in accordance to the specified resolution. We assume this process to be essentially *iterative*: the programmer is involved in a detection-resolution loop, and proceeds by trial and error to fine-tune the specified resolution. This is necessary because it is unlikely that the programmer can correctly specify all resolutions at once, and also because the specification of a particular resolution can have side effects on the interaction space: some interactions can become effective when they were not, new conflicting interactions can appear, etc.

We conjecture that the programmer can, in a finite number of iterations, weight the different tradeoffs and converge to a final solution. Of course, this can imply the realization that two or more aspects are definitely incompatible and can therefore *not* be deployed simultaneously over the application.

3.2 Steps of the Composition Process

First of all, let us assume that the programmer does not take interactions into account when programming the application and the aspects. The aspect processor (be it an interpreter or compiler) consumes such definitions and produces the woven program. During this phase, *detection* of interactions results in a *report* being handed to the programmer. The report includes all kinds of interactions, be they conflictive or not and effective or not. Based on this report, the programmer can reflect upon the situation, fully aware of all the issues at stake (since even non-effective interactions are reported). As a result, *resolution* is specified. As discussed in the previous section, resolution implies that the user explicitly specifies *(a)* dependencies between aspects, *(b)* the visibility of changes made by an aspect to other aspects, and *(c)* the relative ordering of application of the actions of the aspects. We regard such specifications to be are made *declaratively*, and thereafter refer to these as *composition rules.*

In the following run, the aspect processor, now fed with composition rules in addition to (possibly modified) aspects and (normally untouched) base code, proceeds with another weaving pass, whereby dependencies, visibility and ordering are taken into account as specified by the programmer. This modified weaving pass results in another report of interactions for the programmer. The cycles end when the programmer is satisfied with the composition specification.

Technically, this process raises a number of issues: firstly how interactions are automatically detected, secondly how interactions are reported to the programmer, and thirdly what composition rules are and how are they specified. To answer these questions, we now leave the general setting in which we have progressed until now, and consider the particular case of the Reflex AOP kernel, which implements our proposal.

4 Declarative Composition of Structural Aspects in Reflex: General Approach

In this section we progressively dive into our proposal by first introducing Reflex and how structural aspects are defined and implemented in this platform. We then describe the interactions that Reflex is able to detect (Section 4.3) and how they are represented. Treatment of detection, reporting, and resolution is deferred to the following sections.

4.1 Reflex in a Nutshell

Reflex is a kernel for multi-language AOP in Java, that is, an AOP system whose aim is to facilitate the definition and integration of different aspect languages, including domain-specific ones, to modularize the different concerns of a software system. The motivation and requirements for such a versatile kernel were presented in [Tanter and Noyé 2004], and the first global account of Reflex as an AOP kernel in [Tanter and Noyé 2005].

An AOP kernel supports the core semantics of aspect languages through proper structural and behavioral models, easing the task for aspect language designers. A fundamental role of an AOP kernel is that of a *mediator* between different coexisting aspect-oriented approaches; this clearly includes the detection and resolution of interactions between aspects possibly written in different languages. The composition facilities of Reflex were reported in [Tanter 2006a], but focused on the composition of behavioral aspects, with a limited account of structural aspect composition. This paper focuses on the structural part of Reflex, the behavioral model is based on [Tanter et al. 2003].

Note that, as Reflex is a Java AOP kernel, its API is available in Java, and therefore expression of resolution rules will be performed in Java. Aspect languages defined on top of the kernel will most likely opt to express resolution in a more straightforward concrete syntax. As this is a task of the aspect language developer, we will not focus on this topic here.

We choose Reflex as an experimentation platform because it already supports structural aspects based on very powerful reflective

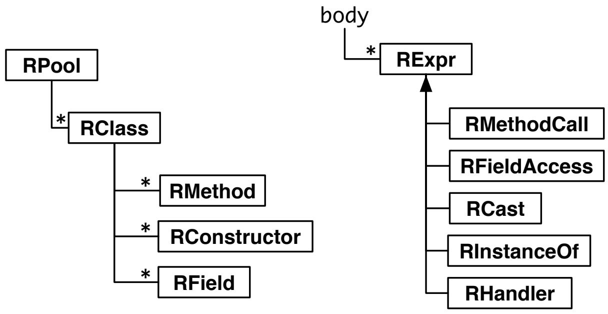

Figure 4: The structural model of Reflex.

model [Tanter and Noyé 2005] that makes it a good fit for the task. In addition, as mentioned above, this work completes the existing support for behavioral aspect composition already available [Tanter 2006a].

4.2 Structural Aspects in Reflex

The abstraction provided by Reflex for defining aspects is that of explicit *links* binding a *cut* to an *action.* A link is therefore the direct correspondent of the primitive aspects we discussed in Section 2.1. Reflex provides both structural and behavioral links, depending on whether the objective is to affect the structure or behavior of an application. An aspect as such is therefore defined as a number of links.

4.2.1 Structural links

A structural link in Reflex (s-link for short) binds a structural cut to a structural action. The structural cut is defined with a *class selector*, algorithmically defining, via introspection, the classes that are affected by the link. The action of the link is defined in a *structural metaobject*, which is a standard Java object that defines structural modifications to classes.

Both class selectors and structural metaobjects operate over a complete reification of the program structure. The structural model is depicted in Fig. 4: an `RPool` object gives access to `RClass` objects, which in turn give access to their members as either `RField`, `RMethod` or `RConstructor` objects (all `RMembers`). The non-abstract members in turn give access to their bodies as `RExpr` objects (with a specific subtype for each kind of expression). The objects are causally-connected representations of the underlying bytecode, offering a source-level abstraction over bytecode. All these are subtypes of `RStructuralElement`.

A class selector is any object that implements a predicate interface used to match `RClass` objects. A class selector can fully introspect the class (down to all its method expressions if necessary) in order to determine whether a class should be matched or not. A structural metaobject can change the definition of a class, by adding structural elements to it. Finally, an s-link is simply defined by associating a class selector with a metaobject.

A structural aspect in Reflex is therefore characterized by the fact that both its cut and actions are *operationally* defined, as opposed to the declarative and limited expressiveness of inter-type declarations in AspectJ. The cut of a structural aspect in AspectJ is restricted to type patterns, and the action is the plain declaration of the members to add, which cannot be parameterized.

4.2.2 Structural correspondence

The structural model of Reflex ensures *structural correspondence* by systematically *hiding all structural changes* made by links to other links.

Structural correspondence, introduced in [Bracha and Ungar 2004], consists in ensuring that the program structure observed via a reflection API corresponds to what one actually expects, rather than including *synthetic* elements added by a processor, compiler, or weaver. For instance, the Java reflection API does not ensure structural correspondence because at runtime one can observe synthetic fields added by the compiler to implement features not directly supported by the virtual machine, such as inner classes.

As discussed in [Tanter 2006a], structural correspondence makes it possible to avoid *unwanted* conflation of extended and non-extended functionalities, as discussed in the meta-helix architecture [Chiba et al. 1996]. Therefore, by default, in Reflex action-cut interactions are always non-effective, because the cut of a link does not see the effects of others. We come back to the visibility issue in Section 7.1, when introducing declarative visibility for per-aspect subjective views on the program structure.

4.3 Interaction Reification

Reflex detects interactions and reports them to the programmer. This section describes the model of structural interactions we have adopted. All interactions detected and reported by Reflex fit in this model (Fig. 5). The purpose of reifying interactions as such is to offer a uniform interface for *interaction report* tools, as discussed later in Section 6.

A structural interaction is represented as an `Interaction` object, instance of one of the three concrete subclasses representing the three kinds of interactions discussed in Section 2.2: `BaseActionInt` for base-action interactions,

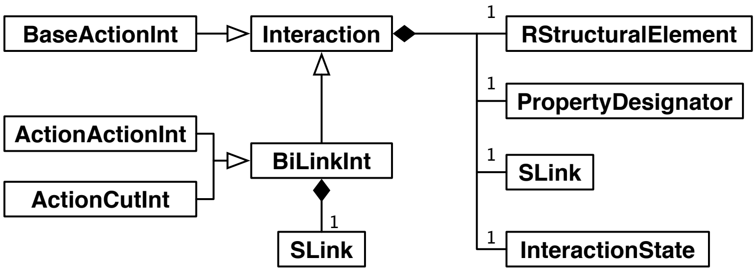

Figure 5: Model of structural interactions.

`ActionActionInt` for action-action interactions, and `ActionCutInt` for action-cut interactions.

An interaction references the structural element (class, method, field, etc.) that is subject to the interaction, as well as a property designator that refers to which property of the element is involved in the interaction. For instance, in an interaction between an aspect looking at the methods of a class for determining its cut and another adding a method to that class, the structural element is the class, and the property designator denotes the "set of methods" property of the class. An interaction also references the s-link that causes it; in the case of interactions involving two links (action-action and action-cut), the second link is also available (see the `BiLinkInt` abstract class). Finally, an interaction has a state, indicating whether the interaction is effective or not. This makes it possible for the programmer to discriminate between action-cut interactions that effectively occur from those that could possibly occur, if the relative visibility and ordering of the involved links were set appropriately (Section 2.4). The interaction state also discriminates between conflicting and non-conflicting interactions, describing the fact that an interaction may come from a compiler exception.

5 Automatic Detection of Interactions

The difficulty of detecting structural interactions depends on their kind (Section 2.2). As already mentioned, base-action and action-action interactions are detected by the bytecode transformer that acts as the weaver because these interactions lead to compilation errors. Therefore detection of these interactions is not discussed any longer. On the other hand, action-cut interactions are much more subtle to detect, precisely because they are not incorrect from a compilation viewpoint. However they may semantically have a great impact[4].

[4] Action-cut interactions do not lead to syntactically erroneous code, such as a class having two versions of exactly the same method. In this sense, they are not detected

5.1 Detecting Action-Cut Interactions

Detecting an action-cut interaction requires knowledge, on the one hand, what structural elements a given aspect effectively *introspects* as part of determining its cut, and on the other hand, what *changes* are performed by other aspects. If one aspect introspects a property that is changed by another aspect, there is a potential interaction.

It is important to note that the issue of the automatic detection of action-cut interactions can be simplified if the aspect system only offers limited and/or declarative means for the cut or the action of a structural aspect (*e.g.* AspectJ and Compose* [Havinga et al. 2006] both have restricted languages for structural cuts and actions). In this work, our objective is to maintain the applicability of Reflex as a *versatile* AOP kernel, therefore we do not accept any alternative that restricts the expressiveness of the kernel. Both cuts and actions are defined operationally in *full* Java over the structural reflective model, as presented in Section 4.1.

Without restricting expressiveness, an alternative that simplifies the detection issue is to require that the aspect programmer *declares explicitly* what an aspect introspects and what it changes. This alternative has the double benefit of simplifying detection, and of ensuring that the kernel detects only interactions that are deductible from the declarations of the programmer; however it imposes a strong burden on the programmer.

In this work we therefore opt for an alternative approach: while maintaining the expressiveness of Reflex, we aim at automatic detection of interactions that does not require any specific declarations from the programmer. Our approach is to use the structural entities themselves as the source of information of what is being observed and changed: during weaving, upon observation and changes, structural entities emit *logic facts* to a logic engine newly integrated into Reflex. Interaction *logic rules* then allow the logic engine to detect interactions. Fact generation and interaction rules are described hereafter. The logic engine is also used in the handling of interaction *resolutions* specified by the programmer, as discussed in Section 7.

5.2 Fact Generation

Two sets of facts are generated by the structural entities themselves in order to keep track of the activity of structural links during weaving: introspection facts (what structural elements are looked at), and intercession facts (what structural changes are performed).

---

straightforwardly by a traditional compiler. Our approach to detecting these interactions is done at weaving time. It would be possible to detect these interactions ahead of time, using static analysis. The difficulty of this task depends on the nature of the structural action language (in our case, a Turing-complete language).

Upon introspection, *i.e.* evaluation of the class selector of a link, structural elements (classes, methods, fields, etc.) generate logic facts indicating that they are being observed by a given link. For instance, suppose a structural link `L1` selects classes that have a field with the `@Persistent` annotation. For the evaluation of its class selector over class `C`, `L1` first accesses the set of fields of the class and then, on each field it accesses the set of annotations of the field and finally, it reads the name of each annotation. This results in the generation of the following facts:

- The `RClass` object representing class `C` generates the fact that `L1` reads its set of fields:
  `readFields('L1','C').`
- Each `RField` object `f` representing a field of `C` generates the fact that `L1` reads its pool of annotations:
  `readFieldAnnotations('L1','C','f').`
- Each `RAnnotation` object `a` representing an annotation of a field `f` of `C` generates the fact that `L1` reads its name:
  `readFieldAnnotationName('L1','C','f','a').`

Similarly, upon intercession, *i.e.* evaluation of the metaobject bound to a link, structural elements generate logic facts indicating the changes being made to them. For instance, if a link `L2` is applied to class `C`, and as part of its action adds the annotation `@Persistent` to its field `f`, then class `C` generates:
`addAnnotationToField('L2','C','C','Persistent','f').`
The above fact includes two classes: the *application class* (*i.e.* the class to which the s-link is being applied) and the *target class* (*i.e.* the class to which the s-link adds the annotation on a field). These two classes need not be the same because, as a side effect of applying to a given class, an s-link can very well perform structural changes on another class (*e.g.* one of its inner classes).

The Reflex logic engine supports similar facts for all possible read and add operations performed on structural elements. We have developed a complete set of structural element classes that generates facts as discussed above.

5.3 Interaction Rules

With the above facts at hand, an interaction is easily detected using the logic engine. An *interaction rule* states that whenever an introspection fact and an intercession fact are related, there is an interaction.

For instance, the interaction rule below states that there is an interaction regarding the annotations of a field `F` in class `TargetCls` between two links `A` and `B`, whenever link `A` reads the set of annotations of field `F` in class `TargetCls`, *and* when link B, applied to AppCls, adds an annotation Annot to the same field F in TargetCls:

```
interactFieldAnnotations(A, B,AppCls, TargetCls, Annot, F) :-
    readFieldAnnotations(A, TargetCls, F),
    addAnnotationToField(B, AppCls, TargetCls, Annot, F).
```

The logic engine includes many interaction rules as above, namely one for each possible kind of action-cut interaction[5]. Section 7.2 gives an operational view on the weaving process, explaining when detection is performed.

6 Reporting Interactions

When an interaction is detected via the logic engine as explained above, a corresponding Java interaction object is created, embedding all necessary pieces of information characterizing the interaction (shown in Fig. 5). Therefore, the result of a detection phase is a *collection of interaction objects*. These interaction objects can be presented in a variety of ways to the application programmer.

In the current implementation, Reflex only uses a text-based interaction report solution. It simply outputs a string representation of all interaction objects (with the option to filter out all resolved conflicts). For instance, the interaction object corresponding to the field annotations interaction presented in the previous section is printed as:

```
Interaction L1-L2 [action-cut/non-conflicting/non-effective]
 -> L1 is reading the set of annotations of field f of class C.
 -> L2 (applied to class C) adds an annotation to field f of class C.
```

The interaction is described with the involved links, its type (action-cut), that it is non-conflicting (no compilation error), and that it is non-effective (meaning L2 is applied *after* L1 reads the set of annotations of f). The rest of the output describes the object of the interaction.

As another example, suppose two links L3 and L4 both add a method of signature int m() to a class C. This action-action interaction is reported as follows:

```
Interaction L3-L4 [action-action/conflicting/effective]
 -> L3 (applied to class C) adds a method int m() to class C.
 -> L4 (applied to class C) adds a method int m() to class C.
```

The interaction is conflicting, meaning that the addition of m by the second link applied could actually not be performed because of a compilation error. It is effective, since all action-action interactions are by definition effective (Section 2.4).

[5] For reference, the list of interaction rules currently used is available at: http://pleiad.dcc.uchile.cl/research/software/reflex/interaction_rules

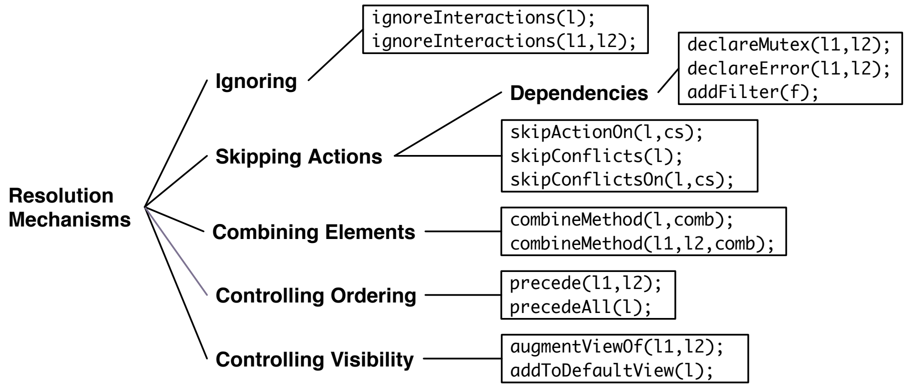

Figure 6: Resolution mechanisms in Reflex.

7 Resolution Mechanisms

Once the application programmer is informed about the various interactions involved in a particular application-aspects setting, resolutions can be declared. In this section we first present the different resolution mechanisms from the viewpoint of the implementer of the aspect language, and then explain in Section 7.2 ends with an explanation of the overall weaving process of Reflex, which supports our proposal.

7.1 Language Implementer Viewpoint

The different resolution mechanisms available to the programmer are summarized on Fig. 6. They are all provided as static methods of the `Rules` class. We briefly discuss them here. Recall that these resolution mechanisms can be reified into the aspect language being implemented using a concrete syntax that the language implementer deems appropriate.

7.1.1 Ignoring Interactions

It is possible for the programmer to state that interactions involving any given link, or interactions between two given links, do not matter and that Reflex should simply ignore them. In the case of non-conflicting interactions, this is no problem, but in the case of conflicting interactions, the consequence is that upon a compilation error, the underlying exception is thrown back to the programmer.

7.1.2 Skipping Actions

The second category of resolution mechanisms results in some aspect action being skipped. There are several ways to achieve this. First, `skipActionOn(l,cs)` declares that the action of `l` should not be applied to classes matched by the `cs` class selector. This can be seen as a mechanism similar to the `global` pointcut restrictor introduced in EAJ [Avgustinov et al. 2006]: a means to further restrict the application of an aspect "from the outside". Then, `skipConflicts` services allow the programmer to declare that if a conflicting interaction (base-action or action-action) occurs, then the action of the responsible link should be skipped (either always or only for interactions related to certain classes).

A sub-category of mechanisms for skipping actions is to declare dependencies, as already introduced in [Tanter 2006a]. One type of dependency is to declare that a link applies whenever another one (*a.k.a.* implicit cut). But more interesting to us here is the mutual exclusion mechanism: stating that a link should not apply if another one does (`declareMutex`). An alternative is to declare that the interaction of two links actually represents an *error* and therefore weaving should not proceed. This is obtained using `declareError`.

For expressing more intricate dependencies between links that are not expressible using the mutex-error mechanism above, Reflex supports lower-level *interaction filters*, specified using `addFilter`. Interaction filters are Java objects that can filter out some links out of a given interaction depending on the links present in the interaction, similar to the combination strategies of JAsCo [Suvee et al. 2003]. Actually, the mutex and error mechanisms are implemented using simple interaction filters.

7.1.3 Combining Elements

When two links add elements with the same signature to the same structural element (*e.g.* two methods `int m()` to a class `C`), it is possible to specify a *combinator*. For instance, a method combinator is an object that, given a method upon which there is a conflict, returns the source code of a method that should be inserted as a combination of the original method on the one hand, and of the new method on the other hand. This mechanism is taken from the composition operator in traits [Schärli et al. 2003]. This feature is at the moment only provided for methods, but one can think of other combinators in the line of Hyper/J [Ossher and Tarr 2001].

7.1.4 Controlling Ordering

Structural links are applied sequentially, in an arbitrary order. If required, the programmer can enforce some ordering constraints, either by stating that a link should be applied before another (`precede`), or by stating that a link should be applied before *all* other links (`precedeAll`). If two links are said to apply before all others, then their relative order is arbitrary, unless a `precede` declaration addressing their relative order is given.

Because by default, as said in Section 4.1, a link does not see the changes made by others, s-link application is typically commutative. However, this is always true from a metalevel point of view (the reifications of program elements), but not always from a base level point of view (the actual code being compiled): it can happen that the code of an inserted method `a` contains a reference to a method `b`: although invisible at the metalevel, the method `b` is required for the proper insertion of `a`. So it can be necessary to enforce ordering for compilation to succeed, or for method combinators to be applied in the desired order. More generally, ordering makes sense when combined with visibility, as discussed hereafter.

7.1.5 Controlling Visibility

When introspecting a class for determining if its cut matches or not, a link only sees what has been declared to be its *view* of the program. By default, a link only sees the original program definition. But it is possible to declare that a link has an *augmented view* of the program, *i.e.* including changes made by other links:

```
(1) Rules.augmentViewOf(l1, l2);
(2) Rules.addToDefaultView(l);
```

Line (1) above declares that `l1` sees all changes made by `l2`. Several links can be given to `augmentViewOf`. Line (2) adopts a different focus, by promoting all changes made by `l` as part of the default view.

To support the subjectivity introduced above, Reflex automatically records the identity of the link affecting a given structural element as a metadata of the element.

Finally, because a link can only *see* changes made to a class by another link *before* actually looking at that class, visibility requires ordering: all visibility declaration always trigger the corresponding ordering declarations (*e.g.* `augmentViewOf(l1,l2)` triggers `precede(l1,l2)`). Note that it is also possible to express *conditional visibility*, *i.e.* visibility that happens only if ordering is separately stated.

7.1.6 Discussion

The presented mechanisms for resolution are always expressed at the level of links or classes. It is indeed possible to go at a finer level of granularity, for instance down to particular members. We have chosen to retain the current granularity for a matter of simplicity, but are willing to refine the API if required.

The above presentation suggests that resolution is expressed in Java, not in Prolog. As said in 4.1, this is because Reflex is a Java AOP kernel. This however still allows the aspect language implementer to define the expression of resolution directly in some other syntax. For example this can be specified in Prolog, or aspect languages with explicit support for aspect composition, like AspectJ `declare precedence`, or languages dedicated to express aspect composition.

7.2 Overall Weaving Process

We now have all the elements required to give an explanation of the overall weaving process of Reflex. A class being loaded first passes through the structural links application phase, before going to the behavioral links setup phase. The reason for the ordering of these two phases is to allow behavioral links to affect join point shadows in structural elements added by structural links (see [Tanter 2006a] for details). Once both phases are complete, the weaver consults the logic engine for all detected interactions and forwards the collection of interaction objects to the interaction report system in use (Section 6).

The structural link application process of Reflex is different from the one originally presented in [Tanter and Noyé 2005]. It has been modified in order to make action-cut interactions possibly effective. Indeed, in the previous version, upon the loading of a class, all s-links were matched against the class to determine the set of applying links, before any was applied. This simply forbids an s-link to see the changes of others, as it cannot evaluate its cut against the modified version of the class[6].

The current process for the s-link application phase is therefore more complex as it implies organizing s-link application in several stages: taking into account the precedence and visibility relations between links, the logic engine is requested for a number of *s-link batches*. A batch contains a number of s-links fulfilling the property that they are *independent*. The application of independent links is commutative: all s-links within a single batch can be matched and applied to the current class in any arbitrary order. On the other hand, if a link `l1` must see the changes made by a link `l2`, then `l1` is put in a batch processed *after* the batch containing `l2`. The independence properties are inferred by the logic engine based on the dependency and ordering specifications.

Within a batch, the process is as follows: all s-links are matched against the current class (*i.e.* their class selector is evaluated to see if they apply), and within the resulting links, mutual exclusion and other interaction filters are applied. Finally, the remaining links are applied.

[6] Note that the structural correspondence issue was dealt with in the context of interactions between the changes made by s-links and the following installation of behavioral links [Tanter 2006a].

8 Related and Future Work

8.1 Related Work

Our general approach on aspect composition is inspired by the work of Douence *et al.* [Douence et al. 2002]: we adopt the proposed framework of *automatic* detection and *explicit* resolution of aspect interactions. However, the present work does not share more with their work, as it is concerned with structural aspects, and [Douence et al. 2002] only focuses on behavioral aspects. Actually, in the area of aspect composition, not much has been done on structural aspects. Most work on aspect composition focuses on behavioral aspects. In AspectJ [Kiczales et al. 2001], base-action and action-action conflicts are reported as compilation errors, while action-cut conflicts are not reported. Furthermore, very little expressive power is given to the programmer to resolve conflicts[7].

Klaeren *et al.* have focused on the issue of validating combinations of aspects [Klaeren et al. 2000]. They use assertions to ensure the correctness of the dependencies between aspects with respect to the specification, focusing on mutually-exclusive aspects. However they do not address means to resolve interactions between aspects. Reflex also covers mutual exclusion, either declaratively or operationally with interaction filters.

JAsCo [Suvee et al. 2003] provides two mechanisms for aspect composition: precedence strategies and combination strategies. Although JAsCo is restricted to behavioral aspects, the above mechanisms are interesting and actually have their equivalence in Reflex, both in the behavioral and structural parts. In JAsCo, an aspect is deployed by specifying a *connector* that determines which *hooks* should be enabled (the cut of an aspect) and which advice should be triggered when the cut is matched. Within a connector that instantiates several hooks, it is possible to specify explicitly the order in which associated advices are executed, leading to fine-grained control on precedence strategies. This is similar to what can be expressed declaratively in Reflex. However, this mechanism of JAsCo works fine only for interacting aspects that are deployed by one connector. On the other hand, Reflex allows precedence declarations to affect any aspect. For other interaction problems that are not solved by means of precedence strategies, JAsCo provides *combination strategies*: a strategy is like a filter on the list of hooks that are applicable at a certain point in the execution. With combination strategies, one can programmatically exclude certain hooks from the current interaction. This is similar to what can be achieved in Reflex with interaction filters. Finally, JAsCo does not automatically report on interactions.

In [Masuhara and Aotani 2006], Masuhara and Aotani discuss issues associated with the interactions between aspect effects and *expressive pointcuts*,

[7] Basically, AspectJ only supports aspect-wide precedence declarations, a solution that has proven insufficient in several scenarios (see *e.g.*. [Brichau et al. 2002, Douence et al. 2002, Tanter 2006a]).

i.e. high-level and/or user-defined pointcuts that specify join points of interest based on more high-level information than mere join point intrinsic properties. This relates to our work because Reflex also supports expressive pointcuts. In the specific context of structural links, expressive structural cuts can be expressed as has been illustrated in Section 4.1. Masuhara and Aotani propose two properties of expressive pointcuts required for aspect interactions, the first of which is directly related to our work: it is stated that *effects of aspects should be visible from the analyses of expressive pointcuts*. In the terminology we used in this paper, this means that structural changes should be visible to the cut of other aspects. This is in the line with the work of Havinga *et al.* [Havinga et al. 2006], which we discuss further below. The SCoPE compiler therefore supports this property by ensuring that the cut of an aspect sees the changes made by others. We conversely adopt an approach in which by default changes are *hidden*, in order to avoid *unwanted* conflation of extended and non-extended functionalities, as discussed in the meta-helix architecture [Chiba et al. 1996]. However, we do *not* hide the fact that there is a *potential* interaction: Reflex detects and reports the interaction, and makes clear to the programmer that the interaction is not *effective* (see first example of Section 6). Only if the programmer desires some changes to be visible to the cut of some other aspects are those changes made visible. This is declaratively stated by the programmer, not automatically decided by the weaver. Declarative aspect composition has also been proposed in [Brichau et al. 2002, Nagy et al. 2004] but they are too restricted to behavioral aspects to be transposable to the case of structural aspects.

In the area of structural aspects, the work of Havinga *et al.* directly relates to ours. In Compose* [Havinga et al. 2006], the approach consists in trying to automatically order structural actions properly, and reject any specification that leads to circularity. The automatic approach to resolution of interactions is interesting, but we rather share the point of view that resolution should be done explicitly, as in many cases, the precise resolution depends on specificities of the considered application [Douence et al. 2002]. As an example, action-cut interactions, although impossible to automatically order, can be taken advantage of rather than resulting in circularity errors. In Reflex, if two aspects have circular dependencies, then the programmer has the full range of choice: choose one ordering or the other, and analyze the result, or consider this circularity an issue and address it by modifying the aspects. In all cases, the programmer is aware of the circular dependency, because *e.g.* an action-cut interaction is reported in both orderings, but can actually declare which ordering is correct.

Lopez-Herrejon, Batory and Lengauer have proposed an algebraic model of aspects seen as program transformation that makes it possible to reason more clearly about aspect composition [Lopez-Herrejon et al. 2006]. They consider both structural advices (introductions, *a.k.a.* inter-type declarations) and behavioral advices (simply called advices). They propose two models for aspects. The first one models aspects as pairs $< a, i >$, where a is the advice part and i the introduction part. In the second model, aspects are modeled as a function $A(x) = a(i + x)$, where x is the program to which an aspect A is applied (+ is the introduction sum, that is, the addition of structural elements). They show that both models differ in terms of the composition they enable. The pair model expresses *unbounded* quantification (*i.e.* the scope of advice covers the entire program), while the functional model expresses *bounded* quantification (*i.e.* the scope of advice extends over a stage in the development of the program). They show that the functional model is more expressive as it can express all compositions of the pair model, and more. Our approach to composition definitely falls into the functional model, as illustrated by the *staged* weaving process of Reflex, which makes it possible for both structural and behavioral advices to have a bounded scope: the scope of advice (in our case both structural and behavioral) is bounded by the actual *view* of that aspect over the aspectual changes made by other aspects.

Mehner *et al.* have proposed a technique for interaction analysis of aspects at the model level [Mehner et al. 2006]. Interactions and dependencies are detected using graph transformation techniques at the level of activities that refine use cases. Although our work is at the program processor level and not at the model level, we share the idea of reporting interactions to system developers in a convenient manner. [Mehner et al. 2006] mentions conflict and dependency matrix as graphical tools to help in the understanding of a system. These visualization techniques are among the many possible interaction reporters we are considering for future work, as discussed in Section 6 and below.

In related areas dealing with structural composition, the method combination approach we have adopted is that proposed for traits in [Schärli et al. 2003]. The generalization of this idea to structural elements other than methods brings us to the general composition operators proposed in [Harrison et al. 2006], whose integration into Reflex as new resolution mechanisms for base-action and action-action conflicts seems both possible and interesting. For these resolution mechanisms, it however seems that offering a dedicated syntax is a must, in order to avoid cumbersome string-based specifications. Extensible concrete syntax for Reflex is on-going work [Tanter 2006b].

Finally, the subjective approach adopted by Reflex, in which aspects have their own *view* on the program structure, which can be declaratively augmented, is, to our knowledge, a distinguishing feature of our work. It enables fine-grained control in the resolution of subtle interactions. Also, the detection and uniform representation of the three kinds of structural interactions (base-action, action-action, and action-cut), in the context of fully expressive cut and action languages, is also, as far as we know, a particularity of this proposal.

8.2 Future Work

We are now exploring a number of extensions to this work. First of all, the handling of behavioral link composition [Tanter 2006a] is currently implemented in Java. The cumbersome implementation of some deductions, *e.g.* for ordering and mutual exclusion, was actually among our main motivations to start integrating a logic engine when working on structural aspect composition. This part should be modified in order to benefit from the logic engine now integrated in Reflex. This should result in a more concise and robust implementation of the existing mechanisms for behavioral link composition.

Once this integration performed, we need to further experiment with the composition process we have presented here, including behavioral aspects, and coming up with an integrated process for both structural and behavioral aspect composition. In order to support this process, it seems crucial to consider appropriate tool support. This means considering an advanced aspect interaction management environment, for assisting the programmer in browsing through detected interactions and declaring their resolution, in an intrinsically iterative manner. Also, recall that because we do not compromise with the expressiveness of the cut and action languages, we generate all introspection and intercession facts that *may* point at an interaction. As a consequence there may be too many reported interactions. An appropriate aspect management environment should help in limiting the cognitive overhead induced by this defensive fact generation.

Finally, since the beginning of this paper, it has been made clear that we only consider aspects whose structural changes consist in *adding* structural elements to a base program. It makes sense to extend this work to other transformations such as direct renaming of structural elements or modification of their other properties.

9 Conclusion

In this work we discussed how to provide full support for the composition of structural aspects in an AOP kernel. We have presented a general analysis of interactions between structural aspects, identifying different kinds of interactions, as well as the corresponding detection and resolution mechanisms. We have then proposed a composition process that involves the programmer in a cycle of automatic detection of interactions and explicit, declarative resolution of these interactions. Finally, we have described a full implementation of the proposed process in Reflex, supporting *(a)* a uniform representation of all the identified interaction kinds, allowing extensible reporting tools to be developed, *(b)* the automatic detection of interactions, including action-cut interactions, based on a logic engine integrated into Reflex, *(c)* the different declarative resolution mechanisms previously highlighted.

On a more global standpoint, this work also illustrates the interest of subjectivity and logic programming in addressing some of the challenges raised by the wider use of aspect-oriented programming. We believe that next generation environments for AOP should consider such advanced mechanisms in order to assist programmers facing the complexity of AOP in the large.

Acknowledgments

We thank Jacques Noyé for his numerous and insightful comments on a draft of this paper, and to Benoit Kessler, Ángel Ñuñez and Rodolfo Toledo for their contribution to the implementation of the features presented here.

References

[Akşit 2003] Akşit, M., editor (2003). *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, Boston, MA, USA. ACM Press.

[Avgustinov et al. 2006] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2006). abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag.

[Batory et al. 2002] Batory, D., Consel, C., and Taha, W., editors (2002). *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. Springer-Verlag.

[Bracha and Ungar 2004] Bracha, G. and Ungar, D. (2004). Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 331–344, Vancouver, British Columbia, Canada. ACM Press. ACM SIGPLAN Notices, 39(11).

[Brichau et al. 2002] Brichau, J., Mens, K., and De Volder, K. (2002). Building composable aspect-specific languages with logic metaprogramming. In [Batory et al. 2002], pages 110–127.

[Bussard et al. 2000] Bussard, L., Carver, L., Ernst, E., Jung, M., Robillard, M., and Speck, A. (2000). Safe aspect composition. In Malenfant, J., Moisan, S., and Moreira, A., editors, *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*, pages 205–210. Springer-Verlag.

[Chiba et al. 1996] Chiba, S., Kiczales, G., and Lamping, J. (1996). Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag.

[Chiba and Nakagawa 2004] Chiba, S. and Nakagawa, K. (2004). Josh: An open AspectJ-like language. In [Lieberherr 2004], pages 102–111.

[Douence et al. 2002] Douence, R., Fradet, P., and Südholt, M. (2002). A framework for the detection and resolution of aspect interactions. In [Batory et al. 2002], pages 173–188.

[Douence et al. 2004] Douence, R., Fradet, P., and Südholt, M. (2004). Composition, reuse and interaction analysis of stateful aspects. In [Lieberherr 2004], pages 141–150.

[Elrad et al. 2001] Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming. *Communications of the ACM*, 44(10).

[Fabry et al. 2008] Fabry, J., Tanter, É., and D'Hondt, T. (2008). KALA: Kernel aspect language for advanced transactions. *Science of Computer Programming*, 71(3):165–180.

[Fabry et al. 2009] Fabry, J., Tanter, É., and D'Hondt, T. (2009). Infrastructure for domain-specific aspect languages: the ReLAx case study. *IET Software*. To appear.

[Glück and Lowry 2005] Glück, R. and Lowry, M., editors (2005). *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, Tallinn, Estonia. Springer-Verlag.

[Gybels and Brichau 2003] Gybels, K. and Brichau, J. (2003). Arranging language features for more robust pattern-based crosscuts. In [Akşit 2003], pages 60–69.

[Harrison et al. 2006] Harrison, W., Ossher, H., and Tarr, P. (2006). General composition of software artifacts. In [Löwe and Südholt 2006].

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

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An overview of AspectJ. In Knudsen, J. L., 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. Springer-Verlag.

[Klaeren et al. 2000] Klaeren, H., Pulvermüller, E., Rashid, A., and Speck, A. (2000). Aspect composition applying the design by contract principle. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, volume 2177 of *Lecture Notes in Computer Science*, pages 57–69. Springer-Verlag.

[Lieberherr 2004] Lieberherr, K., editor (2004). *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK. ACM Press.

[Lopez-Herrejon et al. 2006] Lopez-Herrejon, R., Batory, D., and Lengauer, C. (2006). A disciplined approach to aspect composition. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2006)*, pages 68–77. ACM Press.

[Löwe and Südholt 2006] Löwe, W. and Südholt, M., editors (2006). *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria. Springer-Verlag.

[Masuhara and Aotani 2006] Masuhara, H. and Aotani, T. (2006). Issues on observerving aspect effects from expressive pointcuts. In *Proceedings of ECOOP Workshop on Aspects, Dependencies and Interactions*, Nantes, France.

[Mehner et al. 2006] Mehner, K., Monga, M., and Taentzer, G. (2006). Interaction analysis in aspect-oriented models. In *Proceedings of AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL 2006)*, Bonn, Germany.

[Nagy et al. 2004] Nagy, I., Bergmans, L., and Aksit, M. (2004). Declarative aspect composition. In *2nd Software-Engineering Properties of Languages and Aspect Technologies Workshop*.

[Ossher and Tarr 2001] Ossher, H. L. and Tarr, P. L. (2001). Multi-dimensional separation of concerns and the hyperspace approach. In Akşit, M., editor, *Software Architectures and Component Technology*, volume 648 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer.

[Schärli et al. 2003] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2003). Traits: Composable units of behavior. In Cardelli, L., 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. Springer-Verlag.

[Suvee et al. 2003] Suvee, D., Vanderperren, W., and Jonckers, V. (2003). JAsCo: an aspect-oriented approach tailored for component based software development. In [Akşit 2003], pages 21–29.

[Tanter 2006a] Tanter, É. (2006a). Aspects of composition in the Reflex AOP kernel. In [Löwe and Südholt 2006], pages 98–113.

[Tanter 2006b] Tanter, É. (2006b). An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany.

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

[Tanter and Noyé 2005] Tanter, É. and Noyé, J. (2005). A versatile kernel for multi-language AOP. In [Glück and Lowry 2005], pages 173–188.

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

[Tanter et al. 2008] Tanter, É., Toledo, R., Pothier, G., and Noyé, J. (2008). Flexible metaprogramming and AOP in Java. *Science of Computer Programming*, 72(1-2):22–30. Special Issue on Experimental Software and Toolkits.