Aspects of Composition in the Reflex AOP Kernel

Éric Tanter[*]

DCC – University of Chile
Avenida Blanco Encalada 2120 – Santiago, Chile
etanter@dcc.uchile.cl

Abstract. Aspect composition is a challenging and multi-faceted issue, generally under-supported by current AOP languages and frameworks. This paper presents the composition support provided in Reflex, a versatile kernel for multi-language AOP in Java. The core of Reflex is based on a model of partial reflection whose central abstractions are links: bindings between a (point)cut and an action. Reflex supports the definition of aspect languages through the mapping of aspects to links. We overview the wide range of features for link composition in Reflex—which includes extensible operators for ordering and nesting of links, and control over the visibility of changes made by structural aspects—, illustrating how they can be used to implement various scenarios of aspect composition.

1 Introduction

Aspect-Oriented Programming (AOP) provides means for proper modularization of crosscutting concerns [17]. As a matter of fact, in a typical application, *many* crosscutting concerns can be identified and modularized as aspects. This raises the issue of *aspect composition*, which includes questions such as: how to ensure that aspects are properly composed? Furthermore, since the points where an aspect applies (the *cut* of the aspect) are usually specified intensionally, how can programmers know that two aspects are affecting the same program point?

The issue of aspect composition was first analyzed in [6], where a classification of conflicts between aspects is proposed. Three classes of conflicts are identified: *(a)* inherent conflicts, related to the incompatibility of two aspects, *(b)* accidental conflicts, when two aspects happen to apply at the same program point or have semantical conflicts, and *(c)* spurious conflicts, which are conflicts that are detected whereas they are not actual conflicts. All in all, a number of approaches to aspect composition have been proposed, usually focusing on a particular dimension of aspect composition.

First of all, two aspects that apply to the same program points (text or execution) are said to interact; in other words, the intersection of their cut is not empty. When two aspects interact, there are two possibilities: either they are incompatible, and hence a *mutual exclusion* has to be specified [5, 10, 21], so as to

[*] É. Tanter is partially financed by the Milenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

retract one of the aspects, or to raise an error. Otherwise, if both aspects should be applied, their *order of application* must be specified [5, 10, 31]. If aspects can act *around* an execution point of a program, then the notion of *nesting* appears, typically associated with a `proceed`-like mechanism [20, 31].

Furthermore, one may need to define that an aspect should apply whenever another applies [5, 10] (*aka. implicit cut*), or that an aspect applies onto another aspect [10, 5], for instance using a logging aspect to monitor the effectiveness of a caching aspect (*aka. aspects of aspects*). Finally, in AOP approaches where structural modifications can be done to a base program (*e.g.* adding members to a class), the visibility of these changes to other aspects should be controllable [8].

Finally, the aspect composition problem can be divided in two parts: that of the *detection* of aspect interactions, and that of their *resolution*. SOUL/Aop [5], as well as AspectJ, only address means to *specify* composition, while Klaeren *et al.* [21] focus on means to *detect* interactions. Concrete approaches to detection all deal with conflicts of aspects over a shared program point; being able to detect semantic interactions between two aspects that do not interact from a weaving point of view is to our knowledge not addressed by any proposal, as in the general case it is undecidable. If aspects are expressed using limited action languages, static analysis may be able to detect most semantic interactions (see [15] for an effort in this direction). Using static analysis in presence of Turing-complete aspect languages (at least for the part specifying the actions of aspects) is an open issue. It is also generally admitted that automatic resolution is not feasible; an exception to this is the approach of [15], where the limited expressiveness of the aspect language is used to automatically determine and resolve interactions between aspects. Nevertheless, in a general setting, unless it can be proven that the aspects commute, the resolution of their interaction has to be specified explicitly [10].

We are not aware of any proposal addressing all these dimensions. For instance, AspectJ [20] does not provide any support for mutual exclusion and visibility of aspectual changes, and is limited in terms of aspects of aspects and ordering/nesting of aspects. Furthermore aspect interactions are not detected. Other proposals are thoroughly discussed in Section 7. This paper presents the different mechanisms for aspect composition in Reflex[1], a versatile kernel for multi-language AOP [29][2]. 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.

[1] `http://reflex.dcc.uchile.cl/`

[2] In [29], we only discuss the issue of ordering/nesting of aspects, not the other dimensions. Furthermore, the part on ordering/nesting of this paper includes a number of corrections and improvements over the previously-presented work.

The major contributions of this work are a very flexible solution for ordering and nesting of aspects, and an initial solution for the under-explored issue of how structural changes made by aspects affect other aspects.

In Section 2, we briefly explain the idea of multi-language AOP, and its incarnation in the Reflex AOP kernel for Java. We then discuss the different 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). We then review in Section 7 the literature in the area of aspect composition, highlighting the differences with our work. Section 8 concludes.

2 Multi-language AOP and Reflex

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

2.1 Multi-language AOP

In previous work [28, 29], we have motivated the interest of being able to define and use different aspect languages, including domain-specific ones, to modularize the different concerns of a software system. We have proposed the architecture of a so-called *versatile kernel* for multi-language AOP, and our current Java implementation, Reflex.

An AOP kernel supports the core semantics of various AO languages through proper structural and behavioral models. Designers of aspect languages can experiment comfortably and rapidly with an AOP kernel as a back-end, as it provides a higher abstraction level for transformation than low-level transformation toolkits. The abstraction level provided by our kernel is a flexible model of partial behavioral reflection [30], extended with structural abilities. Furthermore, a crucial role of an AOP kernel is that of a mediator between different coexisting AO approaches: detecting interactions between aspects, possibly written in different languages, and providing expressive means for their resolution.

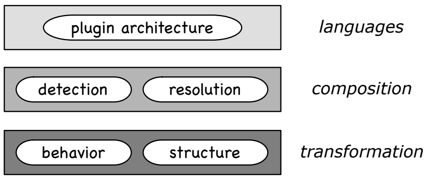

Fig. 1. Architecture of a versatile kernel for multi-language AOP

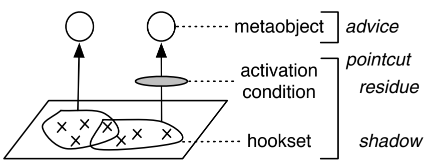

Fig. 2. The link model and correspondence to AOP concepts

The architecture of an AOP kernel consists of three layers (Fig. 1): a transformation layer in charge of basic weaving, supporting both structural and behavioral modifications of the base program; a composition layer, for detection and resolution of aspect interactions; a language layer, for modular definition of aspect languages (as plugins). It has to be noted that the transformation layer is not necessarily implemented by a (byte)code transformation system: it can very well be integrated directly in the language interpreter (VM). As a matter of fact, the role of a versatile AOP kernel is to *complement* traditional processors of object-oriented languages. Therefore, the fact that our implementation in Java, Reflex, is based on code transformation should be seen as an implementation detail, not as a defining characteristic of the kernel approach.

2.2 Reflex in a Nutshell

Reflex is a portable library that extends Java with structural and behavioral reflective facilities. Behavioral reflection follows a model of partial behavioral reflection presented in [30]: the central notion is that of explicit *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. Fig. 2 depicts two links, one of which is not subject to activation, along with the correspondence to the AOP concepts of the pointcut/advice model. Note that our view of AOP is inherently related to metaprogramming: an aspect cut is realized by *introspection* of a program (both structure and execution), and its action consists of behavioral/structural modifications (*intercession*). Reflex does not impose a specific metaobject protocol (MOP), but rather makes it easy to specify tailored MOPs, which can coexist in a given application. This means that one can specify, on a *per-link* basis, the exact communication protocol (which method to call with which arguments) with the metaobject. A detailed case study of supporting the dynamic crosscutting of AspectJ in Reflex can be found in [25].

The aforementioned links are called *behavioral links* to distinguish them from *structural links*, which are used to perform structural reflection. A structural link binds a set of classes to a metaobject, which can both introspect and modify class definitions via a class-object structural model similar to that of Javassist [7]: 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 are causally-connected representations of the underlying bytecode, offering a source-level abstraction over bytecode.

Reflex is implemented as Java 5 instrumentation agent operating on bytecode, typically at load time. The transformation process consists, for each class being loaded, of *(1)* determining the set of structural links that apply to it, and *applying* them, and *(2)* determining the set of behavioral links and *installing* them. The reason of this ordering is discussed in Section 6. During installation of behavioral links, *hooks* are inserted in class definitions at the appropriate places in order to provoke reification at runtime, following the metaobject protocol specified for each link.

2.3 From Aspects to Links

As said above, Reflex relies on the notion of an explicit *link* binding a *cut* to an *action*. Links are a mid-level abstraction, in between high-level aspects and low-level code transformation. How aspect languages are defined and implemented over the kernel is out of the scope of this paper (preliminary elements can be found in [29]). Composition of aspects at the kernel level is expressed in terms of link composition, which is the central matter of this paper.

A simple AspectJ aspect, comprising of a single advice associated to a simple pointcut (with no higher-order pointcut designator), is straightforwardly implemented in Reflex with a link (as in Fig. 2). However, most practical AOP languages, like AspectJ, make it possible to define aspects as modular units comprising *more than one* cut-action pair. In Reflex this corresponds to different links, with one action bound to each cut. Furthermore, AspectJ supports higher-order pointcut designators, like `cflow`. In Reflex, the implementation of such an aspect requires an extra link to expose the control flow information. There is therefore an abstraction gap between aspects and links: a single aspect may be implemented by several links. This abstraction gap is the matter of the language layer, as discussed in [29].

3 Aspects of Aspects

Defining aspects of aspects, *i.e.* aspects that apply to the execution of other aspects, is a feature that can be useful to handle crosscutting in aspects themselves [5, 13, 10]. For instance, a profiling aspect monitoring the efficiency of a caching aspect. Another example is an aspect resolving an accidental semantical conflict between two aspects [6]. Unsurprisingly, Reflex supports aspects of aspects, a feature supported by almost every AOP proposal (*e.g.* the `adviceexecution` pointcut descriptor of AspectJ). 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. Since metaobjects are standard objects, a link can apply not only on the execution of the metaobject methods (similarly to `adviceexecution` in AspectJ), but also to all other operations occuring within the metaobject: field accesses, created objects, messages sent, etc. There is indeed no difference between controlling the execution of a base application object and that of a metaobject.

A distinguishing feature of aspects of aspects in Reflex comes if we consider aspects acting *around* an execution point, for instance a caching aspect. Typically, a caching aspect holds cached values, and when a cache fault occurs, the aspect invokes the original operation via `proceed`. Such a `proceed` is done in Reflex via calling the `proceed` method of an execution point closure (`EPC`) object, which a metaobject can request. If we want to profile the caching aspect to determine the ratio of cache hits/faults, we can define a profiling aspect that matches execution of the caching method, and separately, that of the `proceed` method on the `EPC` object. This definition is not feasible in AspectJ, because `proceed` is a special expression that is not visible to other aspects.

4 Aspect Dependencies

Aspect dependencies can be of two kinds: implicit cut (*"apply A whenever B applies"*) and mutual exclusion (*"never apply A if B applies"*). These dependencies between aspects are mentioned in [5, 10, 21]. In addition, we also consider the case of *forbidden interactions*, an error mechanism to forbid two aspects to interact [6].

4.1 Implicit Cut

An implicit cut is obtained by sharing the cut specification between two aspects: In AspectJ, this is done by sharing pointcuts; in Reflex, by sharing hooksets (pointcut shadows) and activations (pointcut residues). Consider an e-commerce application on which we apply a discount aspect that applies to frequent customers, implemented by link `discount`, and a tracing aspect implemented by the link `trace`. The following ensures that `trace` applies whenever `discount` does (`BLink` stands for *b*ehavioral link):

```
BLink trace = Links.get(discount.getHookset(), <mo>);
trace.addActivation(new SharedActivation(discount));
```

The first line states that `trace` has the same hookset than `discount` (`<mo>` stands for the metaobject specification, not relevant here). The second line adds an activation condition, `SharedActivation`, which ensures that the activation of `trace` is that of `discount`: even if the activation condition of `discount` evolves dynamically, the dependency of `trace` to `discount` is ensured.

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

The above `getSameCut` method is a convenience method equivalent to the previous version. It just hides to programmers the way the implicit cut is realized. Finally, note that an implicit cut by definition implies that both aspects apply at the same points, therefore raising the issue of their ordering/nesting. This is addressed in Section 5.

4.2 Mutual Exclusion

Mutual exclusion between two aspects is obtained in Reflex by declaring that a link should not apply if another one does. As an example, consider a bingo aspect (implemented by a `bingo` link) that is used in the same application as the discounting aspect: every 1000 buyings, a big discount is offered. If a frequent customer happens to be the winner of the bingo, then the standard discount granted to frequent customers should not apply[3]. The following statement specifies that `discount` should not apply if `bingo` does:

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

[3] This example is taken from an EAOP illustration [16, 14].

Following this declaration, Reflex acts differently depending on whether the dependent links are subject to dynamic activation or not. If both links are not activatable (*i.e.* no pointcut residue), the mutual exclusion dependency can be resolved at weaving time, when hooks are inserted in the code. If one of them is indeed subject to dynamic activation, then Reflex postpones the resolution of the dependency to runtime: when control flow reaches a hook shared between mutually-exclusive links, the activation condition of the dominant link (here, `bingo`) is evaluated, and consequently, only one of the two links is applied (`bingo`, or `discount` if `bingo` is not active).

In the face of multiple mutual exclusion dependencies, the current algorithm first sorts out all links which are only dominant and then eliminates dominated links if their dominant is always active, or adds a dynamic condition to the dominated links if their dominant is subject to dynamic activation. At each step, the set of rules that apply is reduced.

For instance, if links `A`, `B` and `C` are interacting and the mutex relations are $mutex(A, B)$ and $mutex(B, C)$, the algorithm first puts `A` in the remaining links set, and removes `B` from the links to consider (supposing `A` is always active). Then, only `C` and `A` remain, and since no mutex is declared between both, `C` is added to the remaining links. The final solution is therefore `A-C`. Now, if `A` is subject to an activation condition, `B` is not removed: rather, it is put in the remaining links, but subject to a dynamic condition on the activation of `A`. At the next step, $mutex(B, C)$ applies. Since the application of `B` depends on that of `A`, `C` would be kept and subject to the activation of `B`. Consequently, at runtime, either `A-C` or `B` result, depending on whether `A` is active or not.

Forbidden Interactions. A particular case of mutual exclusion is when interaction between two aspects should be considered an *error* (*aka.* an inherent conflict [6]). In this case, one does not want to specify which link to apply or not, but rather to raise an exception. This is done in Reflex using `declareError`:

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

Similarly to `declareMutex`, the effect of `declareError` can occur at weaving time if both links are not activatable, or at runtime otherwise. In both cases, a `ForbiddenInteraction` exception is thrown.

5 Ordering and Nesting of Aspects

As previously mentioned, the Reflex AOP kernel follows the general approach advocated by Douence *et al.*, of *automatic* detection and *explicit* resolution of aspect interactions [10]:

- The kernel ensures that interactions are detected, and reported to users upon under-specification (Sect. 5.1).
- The kernel provides expressive and extensible means to specify the resolution of aspect interactions (Sect. 5.2).
- From such specifications, it composes links appropriately (Sect. 5.3).

5.1 Interaction Detection

An aspect interaction occurs when several aspects affect the same program point (execution or structure). Two behavioral links interact *statically* if the intersection of their hooksets is not empty. Still, the cut of an aspect may include a dynamically-evaluated condition (recall Fig. 2): we say that two behavioral links interact *dynamically* if they interact statically *and* they are both active at the same time. Since link ordering is resolved statically (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. Our approach limits the number of spurious conflicts because it is based on the weaving process, which occurs on a by-need basis. In the presence of open systems with dynamic class laoding, two aspects that may theoretically interact for a *given program* (as in the formal approach of [10]) but do not in a *particular run* of that program do not raise detected conflicts.

Two structural links interact if the intersection of their class sets is not empty. We do not discriminate between static and dynamic interaction, because structural links are applied directly at load time. At present our approach for structural link interactions may report spurious conflicts because two links may affect the same class orthogonally. Finer-grained detection of interactions among structural links is left as future work.

Upon interactions, Reflex notifies an *interaction listener*. The default interaction listener simply issues warnings upon under-specification (see [29] for an example), informing the user that specification should be completed. It is possible to use other listeners, *e.g.* for on-the-fly resolution.

5.2 Ordering and Nesting

At interaction points, resolution must be specified. If links are mutually exclusive, specifying their ordering is not necessary[4]. Otherwise, ordering must be specified; this section explains how this is done for behavioral links[5].

The interaction between two before-after aspects can be resolved in two ways: either one always applies prior to the other (both before and after), or one "surrounds" the other [5, 10], although AspectJ only supports wrapping. These alternatives can be expressed using composition operators dealing with sequencing and wrapping. Considering aspects that can act *around* an execution point (such as a caching aspect), the notion of *nesting* as in AspectJ appears: a nested

[4] We deliberately separate the issue of dependencies from ordering/nesting, although mutual exclusion and forbidden interactions could be expressed with the operators explained in this section. The reason is two-fold: first, it is easier and higher-level for the user to declare dependencies as presented in Sect. 4.2; second, it is more efficient for the weaver to "sort out" interacting links before trying to order them.

[5] The case of structural links is simpler because they are always applied sequentially at the time a class is about to be loaded; no nesting is involved.

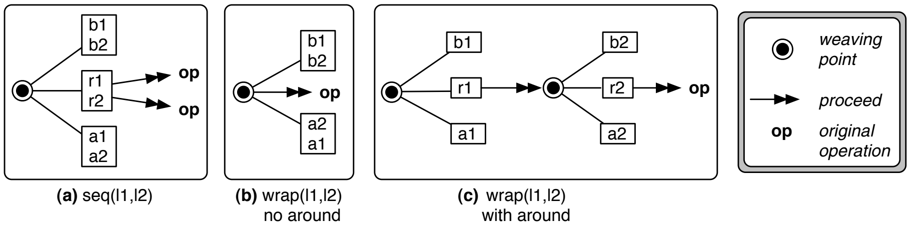

Fig. 3. Ordering and nesting scenarios

advice is only executed if its parent around advice invokes `proceed`. Around advices cannot be simply sequenced in AspectJ: they always imply nesting, and hence their execution always depends on the upper-level around advice [31].

In Reflex, link composition *rules* are specified using composition *operators*. The rule $seq(l_1, l_2)$ uses the *seq* operator to state that l_1 must be applied before l_2, both before and after the considered operation occurrence. The rule $wrap(l_1, l_2)$ means that l_2 must be applied within l_1, as clarified hereafter.

Kernel operators. User composition operators are defined in terms of lower-level kernel operators not dealing with links but with *link elements*. A link element is a pair $(link, control)$, where *control* is one of the control attributes: for instance, b_1 (resp. a_1) is the link element of l_1 for ***before*** (resp. ***after***) control. There are two kernel operators, *ord* and *nest* which express respectively ordering and nesting of link elements. *nest* only applies to *around* link elements: the rule $nest(r, e)$ means that the application of the *around* element r nests that of the link element e. The place of the nesting is defined by the occurrences of `proceed` within r. Sequencing and wrapping can hence be defined as follows:

$$seq(l_1, l_2) = ord(b_1, b_2), ord(r_1, r_2), ord(a_1, a_2)$$
$$wrap(l_1, l_2) = ord(b_1, b_2), ord(a_2, a_1), nest(r_1, b_1), nest(r_1, r_2), nest(r_1, a_2)$$

Fig. 3 illustrates sequencing and wrapping, showing $seq(l_1, l_2)$ with all link elements (a), and the result of $wrap(l_1, l_2)$ first without around link elements (b), and then with around link elements (c). Weaving points are explained later on.

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

```
class Seq extends CompositionOperator {
  void expand(Link l1, Link l2){
    ord(b(l1), b(l2)); ord(r(l1), r(l2)); ord(a(l1), a(l2));
}}
```

```
class Wrap extends CompositionOperator {
  void expand(Link l1, Link l2){
    ord(b(l1), b(l2)); ord(a(l2), a(l1));
    nest(r(l1), b(l2)); nest(r(l1), r(l2)); nest(r(l1), a(l2));
} }
```

The methods `b` (before), `r` (around), `a` (after), `ord`, and `nest` are provided by `CompositionOperator`. The `expand` method, evaluated whenever an interaction between two links occurs, defines a user operator in terms of kernel operators.

Below is an example of a composition rule declared between two interacting aspects: a timing aspect measuring method execution time, and a synchronization aspect ensuring mutual exclusion of methods. Both aspects act before and after method executions. The declared composition implies that the timing aspect measures execution time of methods, *including* the synchronization cost:

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

Another example of composition operator is `Any`: this operator simply states that the order of composition of two given links does not matter (similarly to `commute` in [10]); the kernel is free to compose them arbitrarily. Currently, the `Any` operator is implemented as a `Seq` operator, but this is not something users should rely upon:

```
class Any extends Seq {}
```

Higher-level operators. Users can define higher-level operators based on the building blocks of Reflex. For instance, we can define a variant of `Wrap` that, in addition to the `Wrap` semantics, specifies that the nested link does not apply if the wrapping link is not active. We call this operator `DWrap` (`D` for "dependency"):

```
class DWrap extends Wrap {
  void expand(Link l1, Link l2){
    super.expand(l1, l2); // wrap semantics
    l2.addActivation(new SharedActivation(l1)); // active dependency
 } }
```

5.3 Hook Generation

When detecting link interactions, Reflex generates a hook skeleton based on the specified composition rules, similarly to Fig. 3. The hook skeleton is then used for driving the hook generation process: taking into account how link elements have to be inserted, with the appropriate calls to metaobjects. In order to support nesting of aspects with `proceed`, Reflex adopts a strategy similar to that of AspectJ described in [19], based on the generation of closures.

As mentioned earlier, in order to be able to do `proceed`, a metaobject is given an execution point closure (`EPC`) object, which has a `proceed` method, as well as methods for changing the actual arguments and receiver of the replaced operation. Hence, for each interaction scenario with nesting, Reflex generates closures embedding the composition resolution of the following nesting level, so that calling `proceed` on the `EPC` object results in the execution of the links at the nesting level below. This is done down to the deepest level where `proceed` results in the execution of the replaced operation. The top-level *weaving points* on Fig. 3 represent hooks, while nested weaving points represent closures.

Since previous benchmarks [25] highlighted that executing the replaced operation reflectively implies important performance penalties, we have now adopted the generated stub solution used in AspectJ [19].

6 Visibility of Structural Changes

In the general case, aspects may change both the structure and behavior of a program as a consequence of their actions. Although several AOP proposals – such as EAOP [13, 14, 10, 11], trace-based aspects [12], AOLMP [9, 5, 18], and several others– do not consider structural aspects, languages like AspectJ do (via inter-type declarations). Reflex, as a versatile kernel for AOP, also supports structural changes, as mentioned earlier, via structural links.

As explained previously, aspects rely on *introspecting* the structure of a program to define their cut. Since structural aspects *modify* this structure, the issue of whether structural changes made by aspects are visible to others or not appears. This is a composition issue because if there is only one aspect, there is no problem: the issue arises when considering the integration of several aspects over the same application. This issue is still under-explored in the community.

Consider an aspect adding history to fields, and another aspect making fields persistent: the issue of whether the field added by the first aspect in order to record history should be made persistent appears. In Reflex, the persistence aspect is implemented by a behavioral link, monitoring field accesses; the history aspect, in addition to using a behavioral link for capturing history, makes use of a structural link to introduce a new field in appropriate classes. Therefore, the history field will only be made persistent if the cut of the persistence link actually “sees” that field. For some applications it can make sense to have history fields being persistent as well, but still, those fields may need to be hidden from other aspects.

Default visibility. Reflex applies all structural links *before* behavioral links are setup. This makes it possible for a behavioral link to affect operations related to a member added by a structural link, if so desired. But by default, all structural changes are *hidden*. This makes it possible to avoid *unwanted* conflation of extended and non-extended functionalities, as discussed in the meta-helix architecture [8].

Furthermore, changes done to the program when introducing hooks (for setting up behavioral links) are always hidden. This is motivated by the fact that behavioral changes are conceptually runtime changes: the fact that Reflex operates at load time, by introducing hooks, should be transparent; hence hooks should be hidden. Similarly, infrastructure members introduced by Reflex –such as metaobject references and initialization methods– cannot be observed. This is implemented thanks to a mirror-based structural API [3, 27], which exposes only interface types to users, rather than implementation types as in Javassist: hence Reflex can coordinate visibility of structural elements "behind the scene" (ensuring *structural correspondence* [3, 27]).

Declarative visibility. When introspecting a class for determing matching or not of its cut, a link only sees what has been declared to be its *view* of the program. By default, as we said, 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(persistency, history);
(2) Rules.addToDefaultView(history);
```

Line (1) above declares that `persistency` sees all changes made by `history`. Several links can be given to `augmentViewOf`. Line (2) adopts a different focus, by promoting all changes made by `history` 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. Metadata are stored in a general-purpose key-value property map attached to each structural element, and can be used for many purposes. In particular, it is possible for a link to *force* a new structural element to be always visible (resp. always hidden) by setting a particular property `forceVisible` (resp. `forceHidden`).

The proposed mechanism for controlling the visibility of structural changes already goes beyond existing AOP proposals, in particular AspectJ. However, our approach can still be refined and enhanced, to address more specific and fine-grained conflicts between structural changes.

7 Related Work

Our work on aspect composition in the Reflex AOP kernel is inspired by the work of Douence *et al.* in the EAOP model. It can be seen as an effort to project over a concrete and efficient implementation their formal approach to aspect composition [10, 11]. Among the notable differences is the fact that EAOP does not contemplate structural changes to programs, nor the possibilities of aspects to act around a given execution point.

Klaeren *et al.* have focused on the issue of validating combinations of aspects [21]. 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, as explained in Section 4.2.

JAsCo [26] provides two mechanisms for aspect composition: precedence strategies and combination strategies. 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 using the composition operators. However, this mechanism works fine only for interacting aspects that are deployed by one connector. Also, with respect to around advice however, JAsCo forces the nesting relation, while Reflex lets, at the kernel level, the possibility of having a *sequence* of around advices. For other interaction problems that are not solved by means of precedence strategies, JAsCo provides *combination strategies*: such 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. Again, this is similar to what can be achieved in Reflex; actually the low-level interface in the Reflex kernel is equivalent, except that it works on hook *trees* rather than flat lists, in order to reflect the nesting relation. However, Reflex provides a declarative layer on top of this low-level, programmatic interface, which JAsCo does not. Finally, JAsCo does not automatically report on interactions, and does not address structural aspects.

Nagy *et al.* present a declarative approach to aspect composition [23], considering two types of constraints: ordering and control. The approach for ordering constraints is similar to our kernel-level predicates: the `pre` constraint ressembles our *ord* predicate for indicating precedence. But the issue of aspect nesting (as addressed by *nest*) is not discussed. Control constraints are used to make an aspect depend on the “return value” of the action of another aspect. Although only boolean return values are considered, the approach is interesting. In Reflex, it is expressable in a more flexible manner through activation conditions. Also, Nagy*et al.* introduce two types of constraints, soft ones and hard ones, to be able to express a strong dependency between two aspects, such that one can apply only if the other one did. Mutual exclusion is however not considered. Furthermore, in our proposal, dependencies are a separate notion, although they can be embedded within user-defined composition operators (*e.g.* the `DWrap` operator, Sect. 5.2). Our approach is therefore more flexible in this sense. Finally, they do not address the issue of structural changes to base code.

Brichau *et al.* proposed the use of logic metaprogramming [32, 9] to build composable domain-specific aspect languages [5]. A logic language is used to reason about object-oriented base programs, whose description at the metalevel is done with logic facts and rules. The logic language also serves as the medium in which both aspects and aspect languages are implemented and coordinated, through logic rules in logic modules. Although no aspect-specific syntax is provided, the use of a common logic medium is extremely expressive and allows for the specification of advanced composition strategies. The proposal, called SOUL/Aop, however only considers a static joinpoint model; the more recent AOLMP system Carma [18] is based on a dynamic joinpoint model, but has not gotten to aspect composition issues yet. SOUL/Aop only deals with before/after advices, hence issues related to acting *around* an execution point are not considered; nor are structural aspects addressed. Also, advice weaving in SOUL/Aop is done by inlining advice code at appropriate places, complexifying the support for aspects of aspects. Note that Reflex, as of now, does not offer any real support for composing languages, but just aspects. Conversely, Brichau *et al.* do support composition of languages exactly in the same way as aspects are composed: by combining parameterized logic modules. We are currently exploring language composition alternatives for Reflex, in particular with the MetaBorg approach for unrestricted embedding and assimiation of domain-specific languages [4].

8 Conclusion

We have exposed different dimensions of the multi-faceted issue of aspect composition, and explained the support that the Reflex AOP kernel provides for the same. Reflex supports automatic detection of aspect interactions limiting spurious conflicts; possibilities to express aspect dependencies, such as implicit cut and mutual exclusion; extensible composition operators for ordering and nesting of aspects; the definition of aspects of aspects; and the possibility to control the visibility of structural changes made by aspects. Since Reflex is used as an experimental platform for multi-language AOP, its composition features can be used to handle composition of aspects defined in different aspect languages. The openness of the platform also makes it possible to experiment with new composition operators.

Our experience with supporting declarative aspect composition suggests that an imperative implementation in plain Java may not be the appropriate way to go, as we are facing difficulties in the implementation of some deductions, which would be straightforward using a logic engine. This remains to be explored. Furthermore, our initial solution to composition of structural aspects needs to be extended further, to deal with finer-grained conflicts and resolution schemes.

Acknowledgements. The author would like to thank Jacques Noyé for his detailed comments on a draft of this paper, as well as for his contribution on the body of work on Reflex. Guillaume Pothier, Leonardo Rodríguez and Rodolfo Toledo contributed to the implementation of the features described in this paper. The anonymous reviewers of SC'06 provided very valuable feedback that allowed us to enhance both the presentation and the work hereby presented.

References

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

[2] 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.

[3] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In OOPSLA 2004 [24], pages 331–344. ACM SIGPLAN Notices, 39(11).

[4] M. Bravenboer and E. Visser. Concrete syntax for objects. In OOPSLA 2004 [24]. ACM SIGPLAN Notices, 39(11).

[5] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In Batory et al. [2], pages 110–127.

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

[7] 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.

[8] S. Chiba, G. Kiczales, and J. Lamping. 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, 1996.

[9] K. De Volder and T. D'Hondt. Aspect-oriented logic meta-programming. In P. Cointe, editor, *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection 99)*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272, Saint-Malo, France, July 1999. Springer-Verlag.

[10] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. [2], pages 173–188.

[11] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [22], pages 141–150.

[12] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[13] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, Sept. 2001. Springer-Verlag.

[14] R. Douence and M. Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, École des mines de Nantes, Dec. 2002. 2nd edition, French version published in the Proceedings of "Langages et Modèles à Objets" (LMO'03).

[15] 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.

[16] The EAOP tool homepage, 2001. http://www.emn.fr/x-info/eaop/tool.html.

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

[18] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In Akşit [1], pages 60–69.

[19] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In Lieberherr [22], pages 26–35.

[20] 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.

[21] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. 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, 2000.

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

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

[24] *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, Vancouver, British Columbia, Canada, Oct. 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[25] L. Rodríguez, É. Tanter, and J. Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, Nov. 2004. IEEE Computer Society Press.

[26] D. Suvee, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In Akşit [1], pages 21–29.

[27] É. Tanter. Metalevel facilities for multi-language AOP. In *2nd European Interactive Workshop on Aspects in Software (EIWAS 2005)*, Brussels, Belgium, Sept. 2005.

[28] 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.

[29] 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.

[30] 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).

[31] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, Sept. 2004.

[32] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA 98*, page 112, 1998.