A Versatile Kernel for Multi-Language AOP

Éric Tanter[1]* Jacques Noyé[2]

[1] Center for Web Research, DCC, University of Chile
Avenida Blanco Encalada 2120, Santiago, Chile
[2] OBASCO project, École des Mines de Nantes – INRIA, LINA
4, rue Alfred Kastler, Nantes, France
etanter@dcc.uchile.cl noye@emn.fr

Abstract. Being able to define and use different aspect languages, including domain-specific aspect languages, to cleanly modularize concerns of a software system represents a valuable perspective. However, combining existing tools leads to unpredictable results, and proposals for experimentation with and integration of aspect languages mostly fail to deal with composition satisfactorily and to provide convenient abstractions to implement new aspect languages. This paper exposes the architecture of a *versatile AOP kernel* and its Java implementation, Reflex. On top of basic facilities for behavioral and structural transformation, Reflex provides composition handling, including detection of interactions, and language support via a lightweight plugin architecture. We present these facilities and illustrate composition of aspects written in different aspect languages.

1 Introduction

The existing variety of toolkits and proposals for Aspect-Oriented Programming (AOP) [13] illustrate the fact that the design space of AOP is still under exploration. Low-level toolkits (*e.g.* [8]) can be used to explore the design space and create specific AOP systems, but they require redeveloping an ad hoc software layer to bridge the gap with a proper high-level interface, and they do not address the issue of aspect language design. In this respect, there are proposals of both general-purpose and domain-specific aspect languages. Domain specificity presents many benefits: declarative representation, simpler analysis and reasoning, domain-level error checking, and optimizations [10]. Several domain-specific aspect languages were indeed proposed in the "early" ages of AOP [14, 18, 20], and, after a focus on general-purpose aspect languages, the interest in domain-specific aspect languages has been revived [1, 5, 21, 27].

When several aspects are handled in the same piece of software, it is attractive to be able to *combine* several AO approaches, for instance various domain-specific aspect languages [23]. Yet, combining AO approaches is hardly feasible with today's tools, since the tools are not meant to be compatible with each other: each

* É. Tanter is financed by the Milenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

tool eventually affects the base code directly. This tends to jeopardize correctness when different aspects implemented with different tools interact.

Since most approaches rely upon common implementation techniques, we propose to provide a *versatile AOP kernel*, which supports core semantics, through proper structural and behavioral models. Designers of aspect languages can thus experiment more comfortably and rapidly with an AOP kernel as a back-end, focusing on the best ways for programmers to express aspects, may they be domain specific or generic. The crucial role of such a kernel is that of a mediator between different coexisting approaches: *detecting* interactions between aspects and providing expressive means for their *resolution*.

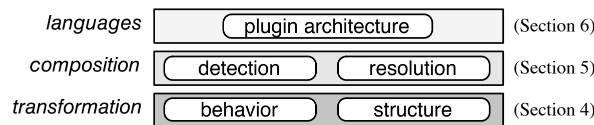

Fig. 1. Architecture of an AOP kernel.

This paper illustrates the evolution of Reflex, originally a system for partial behavioral reflection in Java [26], into an AOP kernel. This experiment gives a first picture of what an AOP kernel may look like and of its benefits. Instead of focusing the discussion on a specific closed proposal, it raises, in a practical manner, the issue of determining what the building blocks of AOP are and how they can be combined in a flexible and manageable way. The proposed 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 interactions; a language layer, for modular definition of aspect languages.

The following section overviews related work, further highlighting the motivation of this work. Section 3 exposes the running example of this paper. We then present the different layers of our AOP kernel following Fig. 1. Section 4 illustrates the core of Reflex as a reflective Java extension, explaining how aspects are mapped to this transformation layer. Section 5 discusses support for aspect composition. Section 6 describes a plugin architecture for modular aspect language support, explaining how plugins are used to bridge the gap between an aspect language and the core reflective infrastructure. Section 7 concludes.

2 Related Work

We now review several proposals related either to multi-language AOP or to extensible aspect languages.

XAspects [23] is a plugin mechanism for domain-specific aspect languages, based on AspectJ [15]. An aspect language is implemented as a plugin generating

AspectJ code, while the global compilation process is managed by the XAspects compiler. XAspects suffers a number of limitations: the compilation process is particularly heavyweight as it requires two full run of the AspectJ compiler; it is unclear whether controlling the visibility of structural changes made to base code is at all feasible; detection and resolution of aspect interactions is not tackled. More importantly, the XAspects compiler provides plugins with the plain binary representation of a program: no higher-level intermediate abstractions are made available to implementors of aspect languages.

Furthermore, although using AspectJ as both the transformation and composition layer of an AOP kernel (Fig. 1) is interesting because of the direct support for expressing crosscutting abstractions, there are several reasons that limit the validity of AspectJ as an AOP kernel. AspectJ is a mature, production-quality aspect language whose practitioner perspective results in limited versatility. In particular, AspectJ is poorly expressive with respect to aspect composition, as will be discussed later, and does not address *detection* of aspect interactions. We rather concur with Douence *et al.* that automatic detection of aspect interactions should be provided [11].

Brichau *et al.* [4] present an approach to building composable aspect-specific languages with logic metaprogramming. Aspect-specific languages are uniformly defined and composed using the same Prolog-like base language: an aspect language is implemented as a set of logic rules in a logic module. This approach provides means not only to compose aspects written in different aspect languages, but also to actually compose languages themselves. A drawback of this approach is that aspect languages do not really shield the programmer from the inherent power of the logic metaprogramming approach: no aspect-specific syntax is provided, aspects are defined in the same logic framework as languages. Finally, this work does not address detection of aspect interactions.

The Concern Manipulation Environment (CME) developed at IBM [9] is a large-scale project aiming to support aspect-oriented software development at any level (analysis, design, implementation, etc.), with respect to any computing environment (programs in various languages, UML diagrams, etc.). The motivation for developing a flexible infrastructure with advanced building blocks to experiment with various AOSD approaches is definitely shared with our work. However, the wide variety of target formats has a serious impact on the concern assembly language: assembly directives are usually specified open-endedly as strings. We rather aim at a higher-level conceptual model to reason about transformation. Finally, detection of aspect interactions is not considered.

Josh [7] is an open AspectJ-like language, which makes it possible to experiment with new means of describing pointcuts and advices. Due to its inlining-based implementation of aspect advices in base code, Josh lacks convenient support for stateful aspects and per-instance aspects. Also, issues related to aspect composition are not addressed.

Finally, *abc*, the AspectBench Compiler is an extensible framework for experimenting with new language features in AspectJ [2]. The spirit of *abc* is similar to Josh, but since *abc* is a compiler, not a load-time tool, it provides a powerful

framework for static analysis. By sticking to AspectJ as the basic language, *abc* presents the inconvenience that both the complexity of AspectJ and that of the *abc* infrastructure (basically a full compiler infrastructure) may be an overkill for simple extensions. As of today, the proposal does not explicitly address the possibility of mixing *different* aspect languages, and aspect composition is still limited to what AspectJ supports.

This work aims to address the limitations highlighted above: there is a need for a versatile kernel for multi-language AOP providing high-level abstractions to implement new aspect languages, and supporting both detection and resolution of interactions between aspects written in different languages.

3 Running Example

The running example of this paper is a multi-threaded program manipulating a buffer, to which three aspects expressed in different languages are applied. The `Buffer` class defines the `put` and `get` methods of an unsynchronized buffer.

First, the buffer is made thread safe using SOM (Sequential Object Monitors) [5], which makes it possible to code separately the scheduling strategy of the buffer. This strategy is implemented in the `BufferScheduler` (discussed in [5]). A small domain-specific aspect language (DSAL) is used to specify that a buffer instance should be scheduled by an instance of this scheduler, as follows: `schedule: Buffer with: BufferScheduler;`

The second aspect is implemented using a general-purpose aspect language, AspectJ [15]. It implements an *argument checking* policy: validation behavior can be attached to join point arguments, either producing exceptions in case of invalid arguments, or simply skipping the invalid call. In our example, we use an argument checker aspect for the buffer, `BufferArgChecker`, which skips invocations of `put` with a null parameter.

The last aspect is used to attach a unique identifier (UID) to objects. This basically consists in adding a private field to hold the identifier, properly initialized, as well as a getter method and the associated interface. It is implemented directly with Reflex. This aspect is provided as a library, with a simple entry point for configuration. To apply it to the buffer, a configuration class[3] is used:

```
public class BufferUIDConfig {
  public static void initReflex(){
    UID.applyTo("Buffer");
} }
```

Now the question is: what happens when all three aspects are applied to the same base program, all affecting the same `Buffer` class? Are calls to the `getUID` method synchronized via SOM, although this is not necessary since it is inherently thread safe? Are rejected calls synchronized as well, or can we make sure that only accepted calls to the buffer are synchronized and scheduled?

[3] Configuration classes are the basic mechanism provided to configure Reflex at start up: their `initReflex` methods are called prior to the execution of the application.

Proposal. Our proposal consists in using an AOP kernel on top of which the different aspect languages are implemented, and having this system report on interactions and offer expressive means for the specification of their resolution. With both SOM and AspectJ available on top of Reflex, applying the three aspects above is done as follows:

```
java reflex.Run -som buffer.som -aspectj BufferArgChecker.aj
                -configClass BufferUIDConfig Main
```

When loading the class `Buffer`, Reflex *detects* the interactions and issues warnings, such as:

```
[WARNING] don't know how to compose SOM and BufferArgChecker.
[WARNING] composing arbitrarily (sequence).
```

The programmer is informed of the unspecified interaction between SOM and ArgChecker. The desired semantics here is to avoid scheduling a request if it is to be rejected (this is correct since validating arguments is thread safe). This can be specified by declaring a *composition rule* stating a *nesting* relation between the two aspects. This declaration can be done in a configuration class:

```
public class CompConfig {
  public static void initReflex(){
    API.rules().add(new Wrap("BufferArgChecker", "SOM"));
} }
```

As we will see in Sect. 5, `Wrap` is a *composition operator* that has the same semantics as precedence in AspectJ. The wrapped aspect (SOM) is only invoked if `proceed` is invoked by the wrapper aspect (BufferArgChecker). If BufferArgChecker rejects a call, it returns without calling `proceed`; hence SOM does not apply, meaning that a reification of the call as a request put in a pending queue until scheduled is avoided. Running Reflex with this composition specification is done by adding `CompConfig` to the list of configuration classes.

4 Overview of Reflex

The analysis of AOP features that led us to the proposal of AOP kernels [25] is concerned with asymmetric approaches to AOP, whereby an aspect basically consists of a *cut* and an *action*: a cut determines where an aspect applies, while an action specifies the effect of the aspect. Depending on the aspect language, specification of the *binding* between a cut and an action may not be decoupled: in traditional reflective systems, the binding between a hook in base code and a metaobject is usually standardized and not customizable, while in a language like AspectJ, it is tied to the action (advice definition).

Reflex relies on the notion of an explicit *link* binding a *cut* to an *action*. As a matter of fact, most practical AOP languages, like AspectJ, make it possible to define aspects as modular units comprising more than one pair cut-action.

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: aspects are typically implemented by several links. This abstraction gap is discussed in more details and illustrated in Sect. 6.

Links are a mid-level abstraction, in between high-level aspects and low-level code transformation. This section overviews and illustrates how such an abstraction is provided and used in Reflex.

4.1 Types of Links

Cuts and actions can be either structural or behavioral. For instance, the UID aspect consists of a selection of structural elements, *i.e.* a *structural cut* (in that case, a set of classes), and a modification of a structural element, *i.e.* a *structural action* (adding several members to a class). Conversely, SOM relies on a selection of behavioral elements, *i.e.* a *behavioral cut* (method invocations), to which a *behavioral action* is associated (reifying calls as requests to be scheduled).

Our Java implementation of the model underlying Reflex is based on bytecode transformation using Javassist [8]. Due to the limitations of the Java standard environment with respect to modifying class definitions, we have to distinguish between two types of links. A *structural link*, termed S-link, binds a structural cut to an action, which can be either structural or behavioral. An S-link is *applied*, *i.e.* its associated action is performed, at load time. A *behavioral link*, called B-link, binds a behavioral cut to an action. A B-link applies at runtime.

We now illustrate structural links (Sect. 4.2) with the implementation of UID, and then show behavioral links (Sect. 4.3) with the implementation of SOM and ArgChecker. Finally, in Sect. 4.4, we discuss how Reflex operates at load time with respect to the different types of links.

4.2 Structural Links

A structural link binds a structural cut to some action (either structural or behavioral). In Reflex, a structural cut is a *class set*, defined intentionally by a *class selector*. For instance, the following class selector defines a cut consisting of the `Buffer` class only:

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

A class selector can base its decision on any introspectable characteristics of a reified class object, *down to the constituents of method bodies* (expressions in a method body are reified if needed). The object model of Reflex wraps and extends that of Javassist: `RClass` objects give access to their `RFields`, `RMethods`

and `RConstructors` (all `RMembers`); both methods and fields give access to their bodies as a sequence of `RExpr` objects.

An action bound to a structural cut is implemented in a *load-time metaobject*, instance of a class implementing the `LTMetaobject` interface. For instance, a `UIDAdder` is a metaobject that applies our UID aspect to a given class[4]:

```
public class UIDAdder implements LTMetaobject {
  static final String UID_FIELD = "private long _uid;";
  static final String UID_GET = "public long getUID(){return _uid;}";
  static final RClass UID_INTERFACE =
                   API.getRClass("reflex.lib.uid.UIDObject");

  void handleClass(RClass aRClass) {
    aRClass.addField(MemberFactory.newField(UID_FIELD, aRClass));
    aRClass.addMethod(MemberFactory.newMethod(UID_GET, aRClass));
    aRClass.addInterface(UID_INTERFACE);
} }
```

Since a load-time metaobject is part of the class loading process, it is a singleton created when the link is defined. A structural link is represented by an `SLink` object. For instance, the following excerpt defines a `uidLink`, which binds the previous `bufferSelector` to a `UIAdder` object:

```
SLink uidLink =
    API.links().addSLink(bufferSelector, new UIAdder(), "UID");
```

Configuration of the UID aspect (Sect. 3) is implemented as follows: the `UID` class holds a single S-link whose class selector is progressively updated by calls to `applyTo`.

4.3 Behavioral Links

This section is both a summary and an update of the model presented in [26]. This model is based on a standard model of behavioral reflection, where *hooks* are inserted in a program to delegate control to a *metaobject* at appropriate places. The particularity of our model lies in the possibility to flexibly group hooks into *hooksets*, and in having explicit and configurable *links* binding hooksets to metaobjects. A hookset corresponds to a set of program points, or static cut (*pointcut shadow* in AspectJ terminology [19]), and the metaobject corresponds to the action to be performed at these program points (*advice* in AspectJ terminology). The link is characterized by several *attributes*; for instance an *activation condition* may be attached to the link in order to avoid reification when a dynamically-evaluated condition is false. An exhaustive discussion of the mechanisms provided for specifying the dynamic part of a behavioral cut (*e.g.* residues) can be found in [24].

[4] The UID code could have been defined in a real class rather than with plain strings.

Defining a B-link. The BufferArgChecker aspect is defined in AspectJ[5]:

```
public aspect BufferArgChecker {
  pointcut checked(): execution(Buffer.put(..));
  around(): checked() { /* check args and possibly skip execution */ }
}
```

This aspect is translated into the following Reflex API calls by the AspectJ plugin (Sect. 6):

```
(1) Hookset putBuffer = new PrimitiveHookset(
      MsgReceive.class, new NameCS("Buffer"), new NameOS("put"));

(2) BLink buffCheck = API.links().addBLink("BufferArgChecker",
      putBuffer, new MODefinition.Class(BufferArgChecker.class));

(3) buffCheck.setControl(Control.AROUND);
(4) buffCheck.setScope(Scope.GLOBAL);
```

First, the cut of the aspect, *i.e.* executions of `put` on a `Buffer` is defined as a hookset `(1)`. A *primitive* hookset is defined by first giving an *operation class*, *e.g.* `MsgReceive`. An operation class represents a kind of operation we are interested in: this corresponds to a join point kind in AspectJ. The set of operations in Reflex is open, meaning the core of Reflex does not support any operation by itself, and can be extended [26]. The definition of the hookset then requires a class selector, which we already presented in the previous section (`NameCS` is a utility that selects classes based on their names); and an *operation selector*, which is a predicate selecting *operation occurrences* (`NameOS` is a utility also doing name-based selection) in program text. Primitive hooksets can be composed in order to obtain more complex hooksets.

The B-link is then defined by associating this hookset to the appropriate metaobject `(2)`. Metaobjects can be obtained either as new instances of a class of metaobjects, or from a factory. `BufferArgChecker` is a metaobject class implementing the desired validation behavior. At this stage the link is defined and operational. Still, we specify some of its attributes: the *control* attribute is set to *around* `(3)`, and since a single instance of the metaobject suffices, the *scope* of the link is set to *global* `(4)`.

As we have just seen, a B-link is represented at load time by a `BLink` object. Because of our implementation approach, a B-link is *set up* at load time and *applied* at runtime. An `RTLink` object represents a B-link during execution: it makes it possible to access and change metaobjects and activation conditions associated to a link at the appropriate level (object, class, or link). An `RTLink` object hence provides a link-specific runtime API for localized metaprogramming. There is a one-to-one relation between a `BLink` and an `RTLink`[6].

---

[5] For the sake of clarity, the advice accesses arguments via the `thisJoinPoint` object rather than via context exposure. Context exposure is briefly mentioned later.

[6] Due to implementation restrictions, the causal connection between both representations is not fully established: runtime changes to the definition of a link will not affect already-loaded classes (some changes are prohibited to avoid inconsistencies).

SOM is implemented similarly: the hookset for SOM consists of the entry points of the public methods of the classes to synchronize. The metaobject (scheduler), is instance-specific, hence the link has scope *object*. Furthermore, since a SOM scheduler needs to act both before and after a method invocation, the control is set to *before-after*.

Context exposure. The particularity of the implementation of SOM is that it makes use of the facilities of Reflex to specify the protocol between a cut and an associated action. This protocol is implemented as a metaobject protocol (MOP). If no custom protocol is specified, the default MOP for a given operation is used [26]. This is usually not efficient because it means reifying information that may not be needed at the metalevel.

In SOM, the scheduler gets control *before* method invocations via invocation of its `enter` method, which receives the name of the invoked method and its arguments, and *after* via its `exit` method, which does not take any parameter. Both `enter` and `exit` are defined in the `som.Scheduler` base class. This specialized MOP is specified as follows:

```
somLink.setMOCall(Control.BEFORE, "som.Scheduler", "enter",
                  new Parameter[]{ nameParam, argsParam });

somLink.setMOCall(Control.AFTER, "som.Scheduler", "exit");
```

The description of parameter generation (such as `nameParam` and `argsParam`) is open and relies on the extended Java language supported by the Javassist compiler, which is both expressive and efficient [8].

Apart from making it possible to program metaobjects without using overly generic protocols, MOP specialization represents a great source of performance improvement. The good performance and scalability of SOM, demonstrated in [5], was obtained thanks to this mechanism. A specialized MOP can be specified at the global level of operations like in traditional reflective systems where the reification of an operation occurrence is standardized and common. But it can also be specified more locally, at the link level, and even at the hookset level. This makes it possible to specialize context exposure at a fine-grained level.

4.4 Process Overview

In order not to modify the standard Java execution environment, behavioral links are *set up* at load time: during the *B-link setup* phase (BLS), hooks, along with necessary infrastructure, are installed in base code at the places indicated by the hookset definitions. Conversely, structural links are *applied* at load time. Since they can influence B-link setup, for instance by inserting a method whose execution is subject to a behavioral cut, the *S-link application* phase (SLA) is carried out before the BLS phase.

This two-phase process is illustrated in Fig. 2. Both phases follow a similar scheme: when a class is loaded, a *selection step* (a diamond in Fig. 2) determines

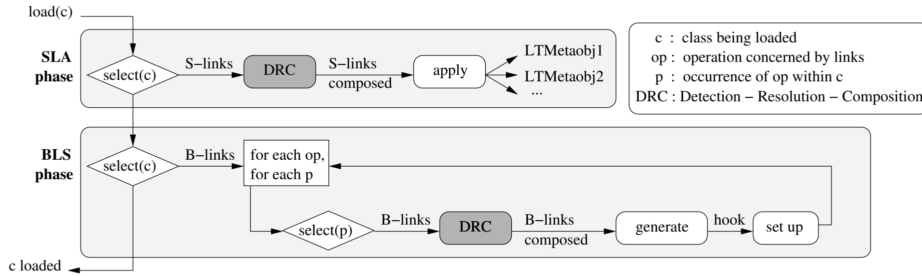

Fig. 2. Reflex operates in two phases at load time; (1) S-link application (SLA); (2) B-link setup (BLS).

the set of links that potentially apply. In SLA, links that select the loaded class are determined, while in BLS, selection goes down to operation occurrences in the class definition. If more than one link potentially apply, a detection-resolution-composition step (DRC in Fig. 2) occurs. Resolution is driven by user specifications; the kernel reports any unresolved interaction. Then links are appropriately composed. DRC is presented in Sect. 5. Finally, S-links are applied (in SLA), and B-links are set up (in BLS) after generating hook code.

5 Link Composition

Aspect composition is a challenging and multi-faceted issue, which is inherently impossible to resolve automatically. Five dimensions related to aspect composition have been identified in the literature, although we are not aware of any proposal addressing them all:

- *implicit cut*: an aspect that should apply whenever another applies [4, 11];
- *mutual exclusion*: an aspect that should not be applied whenever another applies [4, 11, 16];
- *aspects of aspects*: an aspect that applies *onto* another aspect [11];
- *visibility of aspectual changes*: when an aspect performs structural changes, their visibility to other aspects should be controllable [6];
- *ordering and nesting of aspects*: when several aspects apply at the same program point, their order of application must be specified [4, 11, 28].

AspectJ 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. Conversely, Reflex provides initial support for these five dimensions of aspect composition. We hereby only briefly discuss implicit cut, aspects of aspects and visibility of aspectual changes, and pay more attention to mutual exclusion and ordering/nesting of aspects (details can be found in [24]).

An implicit cut is obtained by defining a link whose cut is shared with another link, and aspects of aspects are obtained by links whose cut affects the action (metaobject) of another link.

During the transformation process presented in Sect. 4.4, both the application of S-links and the set up of B-links effectively introspect and modify code raising the issue of whether these modifications should be visible to others. In Reflex, a general-purpose *collaboration protocol* makes it possible to selectively expose or see changes made by other links. By default, Reflex ensures that structural changes made to a class *are not visible* to other links when they introspect the class. This avoids *unwanted* conflation of extended and non-extended functionalities, as discussed in the meta-helix architecture [6]. For our example, the default behavior of Reflex ensures that SOM does not *see* the `getUID` method added by the UID aspect, and hence this method is not subject to scheduling.

5.1 Interaction Detection

Brichau *et al.* [4], as well as AspectJ, only address means to *specify* composition, while Klaeren *et al.* [16] focus on means to *detect* interactions. But, as argued by Douence *et al.*, both detection and resolution of aspect interactions are crucial [11]. Thus we consider them as fundamental features of an AOP kernel.

Our approach follows a detection-resolution-composition (DRC) scheme [11]. The kernel ensures that interactions are detected, and notifies an *interaction listener* upon underspecification. The default interaction listener simply issues warning as shown in Sect. 3, but it is possible to use other listeners, *e.g.* for on-the-fly resolution. The kernel provides expressive and extensible means to specify the resolution of aspect interactions; from such specifications, it composes links appropriately.

An aspect interaction occurs when several aspects affect the same program point (execution or structure). This work is limited to a *static* approximation of aspect interactions. Hence we may detect spurious interactions, *i.e.* that do not occur at runtime. In the process illustrated in Fig. 2, selection steps determine the subset of links that (potentially) apply. If more than one link applies, then there is an interaction. For S-links, there is an interaction when a class being loaded belongs to more than one class set; for B-links, there is an interaction when an operation occurrence in program text belongs to more than one hookset.

In order to support mutual exclusion between aspects, Reflex provides *link interaction selectors*. An interaction selector can be attached to a link, and will be queried whenever the link is involved in an interaction, in order to determine whether it actually applies or not, depending on the other links present in the interaction. Resolving an interaction is hence carried out in two steps: *1)* selecting, within the current interaction, the subset of links that should be applied, and *2)* ordering and nesting the links of the subset. In the following section, we explain how this is supported in the case of B-links. The case of S-links is simpler due to the fact that S-links do not have a control attribute; nesting does not make sense.

5.2 Ordering and Nesting

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 [4, 11].

These alternatives can be expressed using composition operators, *seq* and *wrap*, dealing with sequencing and wrapping. Note that AspectJ only supports wrapping. Considering aspects that can act *around* an execution point (like ArgumentChecker in the example), the notion of aspect *nesting* as in AspectJ appears: a nested 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 [28].

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 ***b**efore* (resp. ***a**fter*) 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 *a**r**ound* 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:

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

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 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 the `CompositionOperator` abstract class. The way user operators are defined in terms of kernel operators is specified in the `expand` method.

Higher-level composition operators can also express mutual exclusion between links. For instance, in Event-based AOP, binary operators like *fst* (resp.

snd) are proposed, expressing that if the left child applies, then the right child does not apply (resp. applies) [12]. *fst* can be implemented by specializing `Seq`, using a link interaction selector stating that `l2` does not apply if `l1` does.

At the kernel level no language support is provided to define rules conveniently: they need to be manually instantiated, node by node (recall the example in Sect. 3). Language support for Reflex configuration (discussed in Sect. 6) can be used to define languages dedicated to composition, or to define languages that include syntactic support for composition. For instance, the notion of precedence of the Reflex version of AspectJ is implemented with `Wrap`.

5.3 Hook Generation

When detecting link interactions, the composition algorithm of Reflex generates a hook skeleton based on the specified composition rules. During this generation Reflex issues warnings whenever composition is under-specified. Users are free to ignore them and let Reflex arbitrarily compose the non-specified parts. The hook skeleton is then used for driving the hook generation process. In order to support nesting of aspects with `proceed`, Reflex adopts a strategy similar to that of AspectJ, based on the generation of closures.

6 Plugin Architecture for Open Language Support

A versatile AOP kernel provides means to modularly define aspect languages, either general-purpose or domain-specific, so that programmers can implement aspects at the level of abstraction that most suits their needs. In Reflex, an aspect language is implemented by a translator to kernel configuration, called a *plugin*. A plugin takes as input an aspect program written in a given language and outputs, either on-line or off-line, the adequate Reflex configuration: links, metaobject classes, selectors, etc., together with calls to the kernel API. The SOM DSAL and (a subset of) AspectJ are the two first aspect languages we have developed for the Reflex AOP kernel[7].

Bridging the Abstraction Gap. A Reflex plugin is typically expected to bridge the abstraction gap between the aspect level and the kernel level. At the kernel level, the main conceptual handle is the notion of *links*. Though making it possible to abstract from low-level details, links are lower-level abstractions than *aspects*. As a result, an aspect is typically implemented by several links.

This abstraction gap can be observed in different scenarios, for instance considering AspectJ support. First, an AspectJ aspect definition may include several pointcuts and advices, plus inter-type declarations; each will be implemented by (at least) one link. Second, the implementation of aspects with higher-order pointcuts requires several links. For instance, if the ArgumentChecker aspect is

[7] All the code (including plugins and the running example) can be obtained from: `http://reflex.dcc.uchile.cl`.

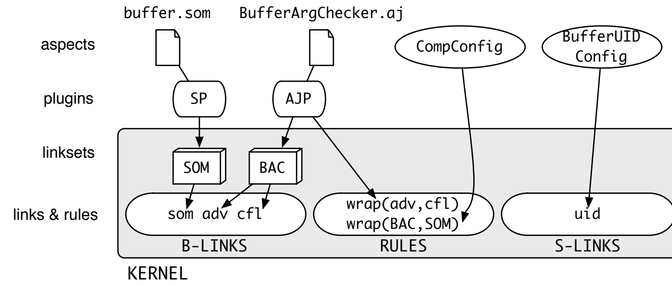

Fig. 3. The running example with the different aspects and their mapping in the Reflex AOP kernel.

extended with a control flow restriction so that nested calls to checked calls are not checked (using `!cflowbelow(p)`), two B-links will be used: the *advice link* for binding the validation behavior, and the *cflow link* for exposing control flow information of the nested pointcut `p`. The cflow link is a before-after link using a simple counter (increased on before, decreased on after). The restriction that the around advice only applies when not below the control flow of an already-checked call is implemented by adding an activation condition to the advice link that checks the value of the counter.

The major issue with this abstraction gap is related to composition. Composition of links related to the same pointcut-advice should be addressed: in the case of control flow above, depending on the order in which the two links are composed, one either obtains the semantics of the `cflow` pointcut designator (first cflow link, then advice link), or that of `cflowbelow` (first advice link, then cflow link).

Links related to the same aspect may also need to be composed. For this issue, AspectJ adopts a syntactic rule whereby advice precedence is defined by the order of definitions in the aspect file. We believe this (implicit) syntactic rule is error-prone (just imagine moving code around). Our approach rather makes such a composition issue explicit and offers more flexible means for its resolution.

When composing aspects that may be written in different languages (implemented by different plugins), the aspect programmer does not care about links. However, all the composition mechanisms of Reflex (interaction notification, composition rules, resolution and generation) work with links, not aspects. In order to support traceability of a link back to its associated aspect-level entity, we introduce *linksets* as a means to group a set of links that are part of the same higher-level conceptual entity.

A linkset is therefore the counterpart, in the kernel world, of an entity in the aspect world. The mapping is defined by the plugin. Reflex accepts linksets in composition rules: they stand for all their links. This semantics is similar to that of AspectJ, where an aspect stands for all its advices in a precedence relation.

Illustration. Fig. 3 illustrates the overall architecture of our running example. The application of SOM to `Buffer` is expressed using the SOM DSAL, implemented by the SOM Plugin (`SP`): it results in the definition of one B-link (`som`), embedded in a linkset (`SOM`). The BufferArgChecker (`BAC`) aspect is expressed in AspectJ, implemented by the AspectJ Plugin (`AJP`): it results in the definition of one linkset `BAC`, encapsulating two B-links, one for the advice (`adv`) and one for the cflow (`cfl`) links, and in the definition of a composition rule `wrap(adv, cfl)` to ensure the `cflowbelow` semantics. The UID aspect is expressed in the configuration class `BufferUIDConfig`, directly adding an S-link (`uid`). And finally, composition between SOM and BufferArgChecker is done in the configuration class `CompConfig`, declaring a composition rule `wrap(BAC, SOM)`.

7 Conclusion

We have proposed an architecture for versatile kernels for multi-language AOP: basic facilities for behavioral and structural transformation, composition handling and language support. The Reflex kernel relies on a reflective model that provides mid-level abstractions to designers of aspect languages: links are a simple abstraction for both transformation and composition. We have exposed the major features of composition support in the Reflex kernel: automatic detection of interactions between aspects and expressive, extensible means for their explicit resolution. A plugin architecture makes it possible to modularly define aspect languages, bridging the abstraction gap between links and aspects. We have illustrated the resolution of interactions between aspects defined in different aspect languages. Future work includes experimenting with more aspect languages in complex scenarios in order to study the scalability of our approach and refine our initial treatment of aspect composition.

Acknowledgments. We thank Leonardo Rodríguez for his work on the AspectJ plugin, and Guillaume Pothier and the anonymous reviewers for their comments.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proc. of ASE 2003*, pages 196–204, Montral, Canada, March 2003. IEEE CS Press.
[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. Technical Report abc-2004-1, The abc Group, September 2004.
[3] D. Batory, C. Consel, and W. Taha, editors. *Proc. of GPCE 2002*, volume 2487 of *LNCS*, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
[4] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In Batory et al. [3], pages 110–127.
[5] D. Caromel, L. Mateu, and É. Tanter. Sequential object monitors. In M. Odersky, editor, *Proc. of ECOOP 2004*, number 3086 in LNCS, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.

[6] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proc. of ISOTAS 1996*, volume 1049 of *LNCS*, pages 157–172. Springer-Verlag, 1996.
[7] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In Lieberherr [17], pages 102–111.
[8] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In Pfenning and Smaragdakis [22], pages 364–376.
[9] The Concern Manipulation Environment website, 2002.
[10] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
[11] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. [3], pages 173–188.
[12] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [17], pages 141–150.
[13] R.E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
[14] J. Irwin, J.-M. Loingtier, J.R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, volume 1343 of *LNCS*. Springer-Verlag, 1997.
[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. of ECOOP 2001*, number 2072 in LNCS, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
[16] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *Proc. of GCSE 2000*, volume 2177 of *LNCS*, pages 57–69. Springer-Verlag, 2000.
[17] K. Lieberherr, editor. *Proc. of AOSD 2004*, Lancaster, UK, March 2004. ACM.
[18] C.V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.
[19] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proc. of CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer-Verlag, 2003.
[20] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, Feb. 1997.
[21] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [17].
[22] F. Pfenning and Y. Smaragdakis, editors. *Proc. of GPCE 2003*, volume 2830 of *LNCS*, Erfurt, Germany, September 2003. Springer-Verlag.
[23] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain-specific aspect languages. In *OOPSLA 2003 DDD Track*, October 2003.
[24] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, Univ. of Nantes and Univ. of Chile, November 2004.
[25] É. Tanter and J. Noyé. Motivation and requirements for a versatile AOP kernel. In *EIWAS 2004*, Berlin, Germany, September 2004.
[26] 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, *Proc. of OOPSLA 2003*, pages 27–46. ACM, October 2003.
[27] M. Wand. Understanding aspects. In *Proc. of ICFP 2003*. ACM, 2003.
[28] 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, September 2004.