Versatile Kernels for Aspect-Oriented Programming

Éric Tanter, Jacques Noyé

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Versatile Kernels for Aspect-Oriented Programming

Éric Tanter, Jacques Noyé

N˚5275

Juillet 2004

Systèmes communicants

ISSN 0249-6399 ISRN INRIA/RR--5275--FR+ENG

INRIA
RENNES

Versatile Kernels for Aspect-Oriented Programming

Éric Tanter*, Jacques Noyé

Systèmes communicants
Projet Obasco

Rapport de recherche n ° 5275 — Juillet 2004 — 27 pages

Abstract: Aspect-Oriented Programming (AOP) is a promising approach to modularizing software in presence of crosscutting concerns. Numerous proposals for AOP have been formulated, some of them generic, others specific to particular concerns. There are commonalities and variabilities among these approaches, which are worth exploring.

Unfortunately, in practice, these various approaches are hard to combine and to extend. This results from the fact that the corresponding tools, such as aspect weavers, have not been designed to be used along with others, although they usually perform very similar low-level tasks. In this report, we suggest to include common functionality into a *versatile kernel for AOP*. Such a kernel alleviates the task of implementing an aspect-oriented approach by taking care of basic program alterations. It also lets several approaches coexist without breaking each other by automatically detecting interactions among aspects and offering expressive composition means.

From a review of the main features of Aspect-Oriented Programming, we present the main issues that the design of such an AOP kernel should address: open support for aspect languages taking care of both behavior and structure, base language compliance, and aspect composition. As this suggests that partial reflection is an appropriate general framework for AOP, we concretize these ideas with a Java AOP kernel based on partial reflection, reconciling in this way reflection and aspect orientation. A case study based on an implementation of the Sequential Object Monitors illustrates the benefits of the approach.

Key-words: programming languages, separation of concerns, reflection, aspect-oriented programming, aspect composition.

(Résumé : tsvp)

* University of Chile and École des Mines de Nantes

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Noyaux versatiles pour la programmation par aspects

Résumé : La programmation par aspects est une approche pleine de promesses pour la modularisation des logiciels en présence de préoccupations transversales. Un grand nombre de propositions pour la programmation par aspects a été formulé. Certaines de ces propositions sont génériques, d'autres spécifiques à certaines préoccupations. Il y a des points communs et variables entre ces approches, qu'il est intéressant d'explorer.

Malheureusement, en pratique, ces différentes approches sont difficilement combinables et extensibles. Les outils correspondants, bien que réalisant des tâches élémentaires similaires, s'avèrent incompatibles. Dans ce rapport, nous proposons d'inclure ces tâches élémentaires dans un *noyau versatile pour la programmation par aspects*. Un tel noyau simplifie l'implémentation de nouvelles approches de programmation par aspects en prenant en charge les transformation de programmes à la base du tissage. Il permet également à différentes approches de coexister sans se gêner en détectant automatiquement les interactions entre aspects, et en offrant des moyens expressifs pour leur composition.

À partir d'une analyse des principales caractéristiques de la programmation par aspects, nous présentons les principaux points auxquels la conception d'un tel noyau doit répondre : support ouvert pour les langages d'aspects prenant en compte à la fois le comportement et la structure, adéquation avec le langage de base et composition d'aspects. Comme cette analyse suggère que la réflexion partielle est un cadre général approprié pour la programmation par aspects, nous concrétisons ces idées avec un noyau pour Java basé sur la réflexion partielle, réconciliant de ce fait la réflexion et la programmation par aspects. Une étude de cas basée sur l'implémentation des *Sequential Object Monitors (SOM)* illustre les bénéfices de cette approche.

Mots-clé : langages de programmation, séparation des préoccupations, réflexion, programmation par aspects, composition d'aspects.

Contents

1 Introduction **4**
1.1 Variety of Models . . . 4
1.2 Compatibility between approaches . . . 4
1.3 Versatile AOP Kernels . . . 5

2 Features of AOP **6**
2.1 Behavioral aspect languages . . . 6
2.2 Structural aspect languages . . . 7
2.3 Interactions application/aspects . . . 8

3 Requirements for a Versatile AOP Kernel **8**
3.1 Open Support for Aspect Languages . . . 8
3.2 Behavior . . . 9
3.3 Structure . . . 10
3.4 Composition . . . 10
3.5 Base Language Compliance . . . 10
3.5.1 Inheritance . . . 10
3.5.2 Concurrency . . . 10
3.5.3 Security . . . 10
3.5.4 Implementation Approach . . . 11
3.6 Interactions application/aspects . . . 11

4 A Versatile AOP Kernel for Java **12**
4.1 Reflex . . . 12
4.1.1 Conceptual model . . . 12
4.1.2 Principles . . . 13
4.2 Behavior . . . 13
4.3 Structure . . . 15
4.4 Composition . . . 15
4.4.1 Detection of aspect interactions . . . 15
4.4.2 Composing aspects . . . 16
4.4.3 Collaboration among aspects . . . 17
4.5 Open Support for Aspect Languages . . . 18
4.6 Base Language Compliance . . . 18
4.6.1 Inheritance . . . 18
4.6.2 Concurrency . . . 18
4.6.3 Security . . . 18
4.6.4 Integration with environment . . . 19
4.7 Interactions application/aspects . . . 19

5 The SOM case study **19**
5.1 SOM with Reflex . . . 19
5.1.1 The SOM configuration language . . . 19
5.1.2 The SOM MetaObject Protocol . . . 20
5.1.3 The SOM runtime API . . . 20
5.2 Compatibility with other approaches . . . 21

6 Related Work **21**

7 Conclusion and Perspectives **23**

8 Acknowledgements **23**

1 Introduction

Aspect-Oriented Programming (AOP) [25, 41] and related modularization technologies for separation of concerns (SOC) [51] have gained wide acceptance. The variety of toolkits and proposals around illustrates the range of possibilities for aspect-oriented programming, either in terms of specification language, binding time, expressiveness, etc. The design space of AOP is under exploration, and each proposal is a fixed point or a restricted region in this space. There are also low-level toolkits that can be used to create ad hoc AOP systems [12, 17, 61]. These toolkits make it possible to experiment with various points in the whole design space.

This work is motivated by the fact that there is a wide variety of models for AO-related programming, either general or domain specific, that are worth experimenting with, and that in general, several approaches cannot be combined simply because they have been designed with a closed world assumption in mind. We propose versatile kernels for AOP that make it possible to use, and experiment with, various approaches, while guaranteeing that approaches do not break each other.

1.1 Variety of Models

There are different conceptual models for programming with aspects. For instance, AspectJ [40] relies on the notions of join points, pointcuts and advices; Event-based AOP (EAOP) [23, 24] uses concepts such as crosscuts, monitors, events and aspects; models from the reflection community rather talk in terms of hooks and metaobjects [53, 59, 63], while the composition filter approach is based on composable method filtering [5]. Interestingly, there are strong links between these conceptual models, as studied by Kojarski *et al.* [43] in the case of reflection and aspect orientation. This comes from the fact that, in the end, they all boil down to semantic alterations of applications written in a base language. A model that has some convincing history in describing semantic alterations is the reflective model for structural and behavioral alterations. However research in aspect orientation has exhibited important behavioral notions related to sequences and nesting of events, as exemplified by control flow and pattern matching of events.

Also, some approaches adopt general-purpose aspect languages [6, 40, 50], while others rely on aspect-specific languages (ASLs, *aka.* DSALs, domain-specific aspect languages). Aspect-specific languages present various advantages, in particular due to the fact that aspects are defined more concisely and more intentionally, since the language is close to a particular problem area. Czarnecki and Eisenecker [16] argue for the benefits of domain-specific approaches: declarative representation, simpler analysis and reasoning, domain-level error checking and optimizations. Several aspect-specific languages were actually proposed in the "early" ages of AOP [38, 45, 48], as well as recently, for instance DJCutter [49] for distributed systems.

The key idea is that the most adequate conceptual model and level of genericity for a given application domain actually depends on the situation: there is no definitive, omnipotent approach that best suits all needs. Furthermore, when several aspects are to be handled in the same piece of software, combining several AO approaches often reveals fruitful [52, 55].

1.2 Compatibility between approaches

Combining several AO approaches seems promising. A positive feedback on a hybrid approach to separation of concerns was reported in [52]. However, in this experiment, specific tools were developed from scratch to fit the experiment. This confirms that combining several AO approaches is hardly feasible with today's tools, since the tools are not meant to be compatible with each other: each tool eventually affects the base code directly, with a "closed world assumption".

If several aspects happen to affect the same program points, they *interact* [22]. If they are implemented through different tools that directly transform the program, the resulting semantics is very likely to depend on the order in which the tools are applied. Interactions among aspects will be silently and blindly handled (Fig. 1a). If the resulting semantics appears to be incorrect, then identifying the interaction and resolving it has to be done manually, if at all possible. This issue presents similarities with the issue of data races in concurrent programming, acknowledged to be one of the hardest errors to debug in the area of concurrency, mainly because the incorrectness of the program is not always visible to programmers and it is very hard to track down to its cause.

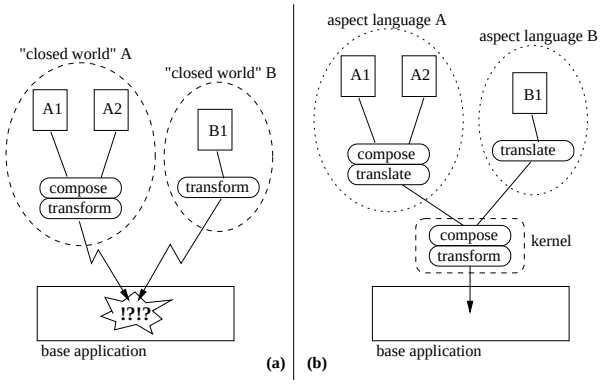

Figure 1: The compatibility issue between AOP approaches.
(a) Different AOP approaches, making a closed world assumption, are applied together: aspect interactions are blindly treated, jeopardizing the resulting semantics.
(b) A common AOP kernel is used as a mediator to detect and resolve interactions: each AOP system only needs to talk to the kernel.

1.3 Versatile AOP Kernels

To sum up the situation, on the one hand, there are many approaches to AOP that are worth exploring and experimenting with, and on the other hand, there is the issue that each AOP language is generally bound to its implementation (as distributed by the creators). This gives rise to the compatibility issue mentioned above. In order to allow AOP to mature, it seems crucial that several approaches can be applied to a wide range of systems and situations, at various scales. Furthermore, efforts should be better focused, without having to "reinvent the wheel" for each new AOP system.

Our claim is that, since the transformation work done by AOP systems is very similar, it can be factored out in a versatile AOP kernel. Each AOP system then talks to the kernel, instead of attacking directly the base application. Only the kernel effectively affects the base application, after having detected the interactions among aspects and let the programmer specify their resolution (Fig. 1b).

An AOP kernel enables a wide range of approaches, from well-established to experimental, to work together without breaking each other. Such a kernel provides core semantics, through proper structural and behavioral models, generic enough to support all needed notions (e.g. cflow, aspects of aspects) in an extensible manner. Designers of aspect languages can then experiment more comfortably and rapidly with this kernel as a back-end, focusing on the best ways for programmers to express aspects, may they be domain specific or generic.

In this work, we are concerned with the study of versatile AOP kernels for a unique base language: we do not aim at multi-language support, or even more ambitiously at any software representation like IBM's CME [32, 36], which attempts to address similarly UML diagrams for instance. We first want to get valuable feedback from an AOP kernel dealing with a single base language. When it comes to illustrating our argumentation, we refer to the language with which we are concretely experimenting, Java.

This work includes the following contributions:

- We motivate the need for a versatile AOP kernel, as a tool for making it possible to combine various AOP approaches. Such a tool would also be useful as a vehicle for further research on AOP.
- From a review of the main features of AOP, we present the main issues that the design of such an AOP kernel should address: open support for aspect languages taking care of both behavior and structure, base language compliance, and aspect composition.
- We show, in the context of Java, how these issues can be solved by building basic aspect weaving and composition facilities on top of a reflective layer, reconciling the flexibility of reflection and the guidance of AOP.
- We describe how these ideas have been used to evolve Reflex [59, 60], an open reflective extension of Java, into a versatile AOP kernel for Java, and illustrate the benefits of such a kernel with the implementation of Sequential Object Monitors (SOM) [10], a library for concurrent programming, developed on top of Reflex.

The rest of this report is structured as follows: Section 2 identifies the different features of an AOP system. From these features, Section 3 draws a list of requirements for a versatile AOP kernel. Section 4 shows how we evolved Reflex as presented in [60] to a versatile AOP kernel for Java. Section 5 presents the case study of the standard implementation of SOM. Section 6 discusses related work, and Section 7 concludes with perspectives.

2 Features of AOP

AOP proposals are characterized by several features. One feature relates to the basic implementation technique (dedicated runtime environment, code transformation, etc.), but this is a non-functional concern for AOP systems, of which AOP kernels take care. We will come back to this point in section 3.

Another feature is the *symmetry* of the approach, as discussed in [31]. We are here interested in *asymmetric* approaches to separation of concerns: approaches where there is the notion of a base application to which aspects are applied, conversely to symmetric approaches to separation of concerns, like hyperspaces [50] or subject-oriented programming [30]. In this setting, the relation between aspects and the base application can be characterized by *aspect obliviousness* and the *binding* between the base application and the aspects.

Aspect obliviousness has been identified as a key property of AOP [25]. It refers to the fact that the base application remains unaware of the aspects that are applied to it. Our point of view on this issue is that obliviousness must be provided, but it should not be considered absolutely necessary. Similarly to the difference between metasystems (systems acting upon other systems) and reflective systems (systems acting upon themselves), it seems too restrictive to prohibit the explicit manipulation of an aspect layer by parts of an application itself subject to aspects.

The binding between the base application and the aspects can be characterized along two lines [53]. The *binding time* refers to the time at which a binding between aspects and the base program can be done. In Java, this can be at compile time (pre/post), at load time, or at runtime (either by the virtual machine or by the JIT compiler). The *binding mode* refers to the rigidity of the binding between aspects and the base program. In particular, it indicates whether a binding can be undone and/or redone dynamically.

Finally, aspect languages can address two types of alterations: behavioral and structural. Aspect languages like AspectJ [40] or Josh [13] include both a behavioral aspect language and a structural one. However, several AOP proposals do not include a structural aspect language.

2.1 Behavioral aspect languages

A behavioral aspect language makes it possible to define aspects that affect the resulting behavior of a base application.

a) the cut language is the language provided to specify the places where aspects affect the base application[1]. A cut denotes a set of *execution points* in a program. Such a dynamic cut can be projected (non-injectively) in the program text to a static cut, which denotes a set of *program points* corresponding to instructions in the program called the *shadows* [47] of the execution points.

Features of a cut language include the expressiveness to refer to both program and execution points (for instance related to control flow), and the possibility to refer to points in aspect programs (to apply aspects to aspects). Cut languages may also allow complex algorithmic cut to be specified [13], or may be tailored for a particular domain [49]. For instance, AspectJ pointcuts are generic (i.e., not domain-specific), can refer to both program and execution points, but can neither refer to aspect program nor describe algorithmic pointcuts.

b) the action language is used to implement the aspect behavior. In the case of behavioral aspect languages, we talk about *behavioral actions* undertaken by aspects, which basically consist in extending and/or modifying the behavior of the base application. The expressiveness of this language may be restricted or designed to fit a particular domain, or may be complete. In addition, the action language may allow the definition of stateful aspects.

c) the binding language is used to specify which aspect should be bound to which cut. Usually this language also makes it possible to specify the kind of control the aspect has over the considered execution points (before, after, and possibly instead of). Many aspect languages do not decouple this language from one

[1] Depending on the proposal, the cut language is either referred to as a *crosscut language* [8, 20] or as a *pointcut language* [13, 41].

of the two above. For instance, in AspectJ, the binding specification is tied to the advice definition: the binding language is merged with the action language.

d) aspect parameterization refers to the possibility of passing context information to aspects. Providing information about the context of an execution point enhances the expressiveness of the action language, and allows for more reusable aspects through genericity. However this has a cost at runtime, especially if the information is passed in a generic manner (e.g., an array of arguments). AspectJ addresses this issue with selective parameter exposition in pointcuts, similarly to the *context exposure* mechanism of Josh.

e) aspect composition may or may not be explicitly part of the aspect language. The Josh language does not provide composition support at all, while the AspectJ language provides a very limited aspect composition language that can only state precedence between aspects. More expressive approaches to composition have been justified [8, 20, 21]. A composition-aware aspect system may provide automatic detection of aspect interactions in order to warn programmers that they should specify aspect composition strategies [20].

f) aspect instantiation and scope are features of the aspect language that specify how aspects are instantiated and what their scope is. AspectJ supports the common aspect scopes: instance, class and global, while Josh does not support per-object aspects. The possibility of discriminating aspect scope with respect to threads may also be provided.

2.2 Structural aspect languages

An aspect language may include a part which is dedicated to perform structural alterations on the base program. This is known as *introductions* or *inter-type declarations*. Such a language allows aspects to perform structural changes to classes, such as adding new members or interfaces. We prefer to identify structural aspect languages as such, and not as an auxiliary feature of aspect languages, to better highlight commonalities and differences between the two kinds of aspect languages. We therefore discuss the same features as for behavioral aspect languages. AspectJ does not deal uniformly with inter-type declarations, while Josh does.

The cut language of a structural aspect language makes it possible to specify where structural modifications should be applied. Whereas a behavioral cut denotes points in the code space, a structural cut denotes points in the data space where data structures reside. In other words, a structural cut denotes program points that correspond to structure *definitions*, not instructions. In the case of structures, there is indeed a bijection between execution points and program points, so it is not necessary to distinguish between them: structural shadows do not make sense. Features of behavioral cut also apply to structural cut. Algorithmic cut can be supported, like in Josh and unlike in AspectJ. The notion of aspects of aspects also applies, considering the possibility for an aspect to perform structural changes on another aspect.

The structural action language provides means to alter the data structures of a program. In a language like Smalltalk [27], such alterations can be done dynamically, while in Java they can only be done statically. Considering a runtime environment that supports runtime structural modifications of classes, a structural action can very well be associated to a behavioral cut (and hence not only to a structural cut). The reverse is not true: a behavioral action cannot be associated to a structural cut (Fig. 2).

	behavioral cut	structural cut
behavioral action	✓	—
structural action	✓	✓

Figure 2: Summary of the possible bindings between structural and behavioral action and cut.

When considering the binding of a structural action to a structural cut, the notion of control (before, after, etc.) becomes irrelevant, simplifying the binding specification. Similarly, the issue of aspect parameterization can also be simplified: since the action consists in modifying selected data structures, in a class-based language, the only information needed is the class subject to modification.

Composition of structural aspects is indeed an issue. This issue has however not been explicitly considered by existing proposals related to aspect composition, which focus on behavioral aspects [8, 20, 21, 22].

Aspect instantiation and scope are meaningful features for structural aspects only when considering runtime structural actions. If not possible, like in Java, then a structural aspect is applied statically, hence a single global instance suffices.

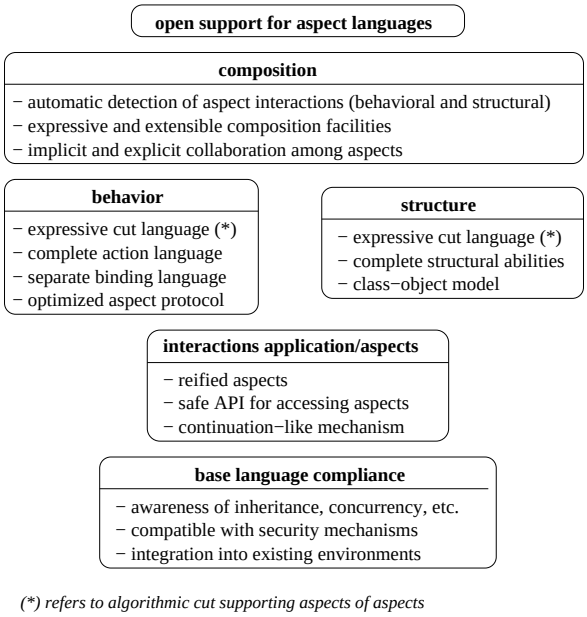

Figure 3: Summary of identified requirements for a versatile AOP kernel.

2.3 Interactions application/aspects

An independent feature of AOP proposals relates to the interactions between an application and the applied aspects. Such interactions need to be characterized in both ways: interactions from aspects to the application, and interactions from the application to aspects.

The action language determines the possible interactions between the aspect and the base application. For instance, it may include a continuation-like mechanism, like AspectJ's `proceed` keyword that entails the interrupted operation to be resumed in the case of around advices.

Interactions between the application and aspects may be provided in systems where aspects are runtime entities as such, i.e. aspects are *reified*. If aspects are inlined within application code, the application cannot explicitly interact with them. Conversely, reified aspects are made accessible through an API for explicit access by the base application. For instance, in AspectJ, the object representing an aspect `Foo` can be accessed with `Foo.aspectOf()`. An access API may be limited to read access, or may make it possible to dynamically change aspects. Changing aspects may relate to changing actions or cuts, depending on which parts are reified. For instance, in AspectJ, only advices are reified, and are not changeable. Conversely, Steamloom [7] fully reifies aspects, making it possible to access cuts at runtime.

3 Requirements for a Versatile AOP Kernel

All the features exposed in the previous section represent the main *variabilities* among the family of aspect languages and systems. The objective of a versatile AOP kernel is to support the range of aspect approaches by supporting these variabilities. In this section we extract various requirements for AOP kernels in a general setting, summarized in Fig. 3.

3.1 Open Support for Aspect Languages

An AOP kernel makes it possible to use particular AO approaches for handling particular aspects. The family of aspect languages being open-ended –all the more as it includes generic and specific aspect languages–, the kernel must provide *open support* for aspect languages.

The language and underlying conceptual model of an AOP kernel (hereafter L_0) has to be general enough to handle all the variabilities presented in section 2. We have identified three main concerns for L_0, namely behavior, structure and composition (Fig. 4). For behavior and structure, both introspection and intercession should be supported: introspection deals with program *analysis*, while intercession deals with program

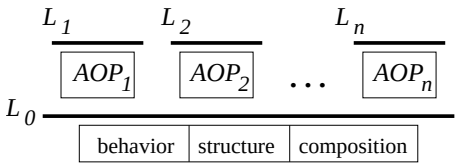

Figure 4: Elements of the AOP kernel approach.
Each AOP system offering an aspect language L_i is implemented as a translator AOP_i to the common kernel language L_0. The kernel language has 3 main parts: one for behavioral manipulation of the base application, one for its structural manipulation, and one for specifying composition.

transformation. For composition to be manageable by an AOP kernel, each aspect language L_i must not be implemented as a code transformer directly affecting base application code, but rather as a *translator* from L_i to L_0 (AOP_i). Behavioral and structural parts of L_0 must be an adequate back-end for any aspect language, while the compositional part has to ensure that *a)* aspect languages can express their composition facilities with it, *b)* interactions among aspects, possibly defined with different languages, are detected and can be resolved, *c)* collaboration between aspects is correctly handled. All these issues are discussed in the remainder of this section.

3.2 Behavior

The kernel language must provide an expressive behavioral cut language, supporting aspects of aspects and complex algorithmic cuts. The dynamic cut should support behavioral notions highlighted by research in aspect-oriented programming (e.g., control flow). This support ought to be provided by the conceptual model at a generic level in order to allow various notions to be implemented in various ways.

For instance, different variants of control flow can be designed: control flow can be exposed as a simple call stack depth counter, or as an event stack, offering the various elements of the call stack for introspection. This allows for more expressive control flow conditions in the cut language. Furthermore, control flow, as considered in most proposals (e.g., AspectJ and Josh), is only about *nesting* events, not about their *sequences* generally speaking. Research on event-based AOP and trace-based aspects [20, 21, 22, 23, 54, 26] has justified the benefits of being able to define aspects that apply depending on the execution history. Douence *et al.* study the formalization of such aspects [20, 21, 22, 23]. Sereni *et al.* propose control flow as regular expressions on the call stack [54]. Filman *et al.* further extend the design of a language of events in order to fully express relationships among events, such as timeframe of occurrence [26].

To handle all possible action languages, the kernel action language should support stateful aspects, so that they can maintain information across execution. Since the kernel action language should also be complete, using the base language is the logical choice.

The binding language deserves special attention. As mentioned before, the binding language of AspectJ is merged with the action language: the advice body of an aspect is tied to the binding to a pointcut. This brings performance benefits, since only selected parameters are exposed by a pointcut to an advice. Also, from an ease-of-use perspective, it has the benefit of almost hiding to the programmer the vertigo of writing metaprograms. However, from a software engineering perspective, this limits the possibilities of reusing a given advice in a different context. Looking back at the history of runtime metaobject protocols in reflection, the position is the opposite: the binding language is tied to the cut language. The set of reified information is both rich and standardized, determining the actual *protocol* of metaobjects (hence the term "metaobject protocol", MOP). Metaobjects are highly generic and hence reusable, but costly and complex to write.

Both approaches have their advantages. For building middleware or other kind of infrastructure software, the genericity and high reusability of metaobjects is of great value. For localized AOP, the non-genericity of aspects is not a problem, it is even a plus, simplifying the task of aspect programming. This duality of approaches with respect to the binding language naturally calls for a versatile AOP kernel that keeps the binding language *separated from both* the cut and action languages, and that provides mechanisms for specializing and optimizing the information bridge between execution points and aspect bodies, which we refer to as *aspect protocol*.

3.3 Structure

An AOP kernel should in particular offer an expressive cut language supporting algorithmic cut and aspects of aspects. For structural actions, a complete set of structural transformation abilities has to be provided. This part consists in offering full structural reflection on the application, and letting the different aspect languages use (part of) these abilities. For class-based languages, the intuitive structural description of programs is the class-object model [62], which represents a program by class objects aggregating member objects (fields, methods). This model follows the seminal model for structural reflection developed in Smalltalk-80 [27].

3.4 Composition

As said in the introduction, two aspects are said to *interact* if their actions must be executed at the same execution point [22]. As argued in [20], the *resolution* of aspect interactions ultimately depends on the application semantics and hence cannot be decided automatically. However, interactions can be *detected* automatically. Hence, the AOP kernel should detect interactions and warn the programmer, so that composition strategies can be specified to resolve the interactions.

Composition should be supported in an expressive and flexible manner. A poorly expressive composition language, like in AspectJ, where only aspect precedence can be specified, is not sufficient to handle complex interactions between aspects [8, 20, 21]. Composing aspects does not solely refer to specifying the order in which they apply, but also to possibly condition their application to the presence and application of other aspects. Composition issues relate to both structural and behavioral changes.

Finally, in AOP it is common practice to introduce structural properties (such as an implemented interface) that may then be visible to other aspects. Conversely, some structural changes may be totally local to a given aspect and should not be exposed to others. This implies that a *collaboration protocol* should be provided to control the visibility of structural changes among aspects. Collaboration between behavioral changes is by definition implicit: an aspect does not see another aspect unless its cut affects it (if aspects of aspects are supported).

3.5 Base Language Compliance

An AOP kernel should support the various semantic elements of its base language. For instance, Java is a class-based language, offering inheritance. But is has also been designed for concurrent and distributed programming, and supports security policies. These elements are usually underestimated in the various proposals and ad hoc toolkits, or simply left aside. However, they can have a non-negligible impact on the design of the kernel and its features. We therefore discuss this issue for inheritance, concurrency and security. Persistence and distribution are not addressed in this report. A discussion of the impact of distribution over the design of a cut language can be found in [49]. Finally, the issue of the implementation approach and integration of the kernel into existing environments is discussed.

3.5.1 Inheritance

A Java AOP kernel is expected to behave well with respect to the interaction between inheritance and aspects. The main concern in this regard comes from the fact that aspects are introduced by modifying class definitions, and that inheritance implies that subclasses inherit the structure and behavior of their superclasses. Conceptually, since aspects are dedicated to handle concerns that crosscut the class modularization, their scope does not necessarily follow that of the inheritance hierarchy. Hence an AOP kernel should make it possible to declare if the cut of an aspect applies to subclasses or not, and ensure consistency of the resulting semantics.

3.5.2 Concurrency

A Java AOP kernel must also be usable in concurrent environments. Aspects can be subject to concurrency, and they may as well be used to control concurrency in an application. This entails that the visibility of an aspect with respect to threads (global or local) should be specifiable, and, in the case of dynamic aspects, their initialization should be thread-safe.

3.5.3 Security

Java offers a simple security mechanism based on policies. There is also a dual relation, like for concurrency, between aspects and security. Much work has been done on implementing security policies with reflection or

aspects [64, 1, 19]. But the reverse is indeed crucial: aspectizing an application must not break its security properties. Vayssière *et al.* studied this issue in the case of a simple Java runtime metaobject protocol [9, 11]. A first issue is to ensure that using a metalevel does not tamper with properties of the base application. A second issue is to devise security policies, compatible with the existing Java policy mechanism, to protect the base application by restricting the actions available at the metalevel (such as changing the receiver of a method call or its arguments).

The fundamental point is the necessity of keeping *metacode separated from base code at runtime*. This is required because the security mechanism of Java relies on the call stack to dynamically compute the permissions associated to a call [28]. Hence, if metacode is inlined within base code, it gets exactly the same permissions as base code. The solution is that only infrastructure code should be inserted in the base application. Infrastructure code is assumed to be trusted –and hence can get the same permissions as base code– since it is generated by the reflective (or aspect) system and solely delegates to the possibly untrusted metaobject (or aspect) code.

3.5.4 Implementation Approach

There are two main implementation approaches to AOP systems, one that consists in extending or modifying the runtime environment of the language, and one that consists in transforming code and leaving the runtime environment intact. To fully support dynamicity, an AOP kernel should be closely integrated into the language environment, in particular into the runtime environment.

In the context of Java, this means that the AOP kernel should be provided by the Java Virtual Machine (JVM) itself. However, the abilities of standard JVMs with respect to behavioral and structural intercession are limited. Hence, experimenting with an AOP kernel at the VM level requires working on a dedicated environment. To be compatible with standard Java environments, we decide to slightly limit the dynamicity supported by our kernel and thus adopt a code transformation approach. We feel that this choice can be beneficial, at least in a first phase, to study AOP kernels in various settings, since it allows simpler and more widespread experiments to be carried out. If AOP kernels turn out to be of practical interest for the Java community, then VM support for AOP kernel services should be considered.

In order to be used as a weaver, an AOP kernel should first be parameterized by a set of aspect languages:

$$(1)\ kernel(\{L_i\}) \Rightarrow kernel_{\{L_i\}}$$
$$(2)\ kernel_{\{L_i\}}(application, aspects) \Rightarrow application_{aspectized}$$

(1) The AOP kernel is parameterized by a set of supported aspect languages ($\{L_i\}$). A language L_i is implemented as a translator from L_i to L_0. Depending on the kernel, parameterization may or may not be available dynamically to incrementally update the set of supported aspect languages. (2) The specialized AOP kernel then acts as a *weaver*, producing the aspectized application from the application and the various aspects (written in any of the aspect languages belonging to $\{L_i\}$). The weaver is conceptually a composition of the translators of the $\{L_i\}$ languages.

For this approach to be acceptable from an efficiency viewpoint, the kernel must generate an aspectized application that performs approximately as well as it would in the case of a specialized AOP system. This means that the kernel should not generate an aspectized application where everything is dynamic, but instead should be able to use *staging*, fixing some concerns statically in order to enhance performance, in case the approach does not require dynamicity. For instance, regarding aspect composition, it can be resolved statically [8, 40] or dynamically [20, 59]. The AOP kernel should allow composition issues to be resolved statically (at weave time) thanks to an extensible set of composition operators, and also let the possibility of using composition frameworks for dynamic aspect composition. Staging in the AOP kernel can thus be seen as a tradeoff between choices fixed at weaving time and others left open at runtime. Also, the AOP kernel should possibly be used offline as a post-processor, or online as a special class loader.

3.6 Interactions application/aspects

Several requirements identified beforehand indicate that an AOP kernel should not inline aspect code within application code, but rather adopt a model where aspects are runtime entities separated from the objects they influence, i.e. aspects should be *reified*: reifying aspects makes it possible to expose their actions to other cuts, hence supporting aspects of aspects; aspects being runtime entities can also maintain some state during execution; reified aspects are compatible with stack-based security mechanisms. Furthermore, the incurred performance penalty is far from obvious in the context of ever-improving dynamic compilers [34].

Finally, for an AOP kernel to fully support interactions between applications and aspects, reifying aspects is also mandatory. In particular, a continuation-like mechanism is necessary in order to support approaches that allow aspects to *replace* operation occurrences. With respect to the interactions with aspects from within an application, it should be possible to access reified aspects in order to explicitly interact with them, and also to change them. However, the access API supported by the kernel should be safe: it must be possible to specify that an aspect *cannot* be changed during execution, as well as to impose some restrictions for dynamically replacing aspects. For instance, type restrictions can be used to guarantee that an aspect protocol will not be broken by replacing an aspect action with another one.

4 A Versatile AOP Kernel for Java

In this section we present the evolution of Reflex as presented in [60] towards a versatile AOP kernel for Java. After giving an overview of the underlying conceptual model and principles of the Reflex AOP kernel, we address the requirements identified in the previous section (Fig. 3). We shall start with the central issues of handling behavior and structure, before elaborating on the other issues.

4.1 Reflex

4.1.1 Conceptual model

In the previous section, we have identified the three main concerns for the kernel language: behavior, structure, and composition. The kernel language has to be general enough to support various approaches, and must support the specificity required by a single approach. We suggest that reflection is the appropriate general framework for AOP kernels, provided that the reflective behavioral model supports *a)* the notion of crosscutting metaobjects making it possible to link several execution points to the same metaobject, and *b)* a high degree of selectivity and specialization. Our approach is therefore based on *partial reflection* [37].

In [60], we expose the model of hooksets for *partial behavioral reflection*, which bridges the gap from classical behavioral reflection to AOP, by explicitly supporting crosscutting metaobjects. Let us remind that this model presents three layers (Fig. 5): on the first layer, hooksets are composable sets of hooks, that is, interception and reification points in the base program. On the third and topmost layer lie metaobjects, which control the execution at hooksets. Metaobjects typically implement aspect behavior (i.e., reify aspect actions). Between the base level and the metalevel lies the *link*, made explicit in this model. One of the main features of the link is to be activatable. A deactivated link makes it possible to avoid useless reification (i.e., activation is checked dynamically before reifying any information). Finally, the link is characterized by other attributes (such as scope –object, class, hookset–, and control –before, after, replace–).

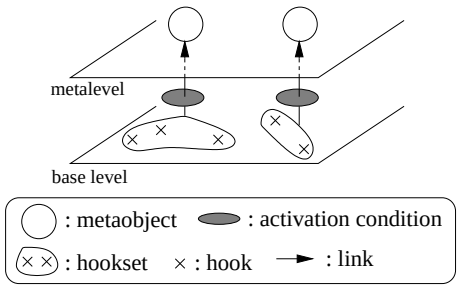

Figure 5: The hookset model for partial behavioral reflection.

Reflex, as presented in [60], is a portable Java implementation of the hookset model, operating on bytecode, which also provides high opportunities for configuration and specialization. Since the core of AOP is about behavioral changes to applications, we considered Reflex a good starting point for building an AOP kernel. To evolve Reflex towards an AOP kernel, we most importantly needed to add structural abilities, advanced composition facilities, and open language support.

To support structural aspects, we have extended Reflex with a notion of *structural links*. A structural link binds a structural cut (a set of classes, say, a *class set*) to a structural action, implemented by a *structural*

metaobject. In the rest of this section, we shall differentiate structural and behavioral elements (links, metaobjects, etc.) with respectively *s-* and *b-* prefixes: e.g, s-links and b-links. Abilities to transform classes structurally rely on Javassist [12, 14], a powerful library for load-time structural reflection.

AO concept	realization in Reflex	
	behavior	***structure***
action	b−MO	s−MO
binding	b−link	s−link
static cut	hookset	class set
dynamic cut	hookset + activation	–
parametrization	MOP descriptors	*fixed:* class object
instantiation	MO instantiation	*fixed:* lazy
scope	MO scope	*fixed:* set
static composition	link composition	link composition
dynamic composition	b−MO composition	–

Figure 6: Mapping of AOP concepts to their realization in the Reflex AOP kernel.
("MO" is used as an abbreviation for "metaobject".)

The conceptual model of our AOP kernel is therefore the hookset model for the behavioral part and the class-object model for the structural part. Composition support is inspired from the work of Douence et al. [20, 21]. The mapping between AOP concepts as used since the beginning of this report and reflective ones used in the rest of this section is given in Fig. 6.

4.1.2 Principles

Since Java does not support dynamic modification of class definitions, Reflex operates differently for s-links and b-links (Fig. 7). A structural action can only be bound to a structural cut via an s-link. Since s-links are applied statically, at load time, their interface, scope and instantiation mode is fixed. When a class is loaded, if it is determined to be part of a class set, the corresponding s-metaobject is applied. Conversely, b-links are only *installed* at load time, while they are applied at runtime. Indeed, when a class is loaded, hooks and other necessary infrastructure code are inserted into it. B-metaobjects only act during execution.

Reflex operates in two phases at load time: in a first phase, s-metaobjects that need to be applied to a class being loaded are determined, possibly composed, and applied to the class. In a second phase, b-links that affect the class being loaded are determined, composed at each interaction point if any, and the corresponding hooks are generated and inserted in the class. An overview of this 2-phase process is given in Fig. 8, details are discussed in the remainder of this section.

4.2 Behavior

In our model, points in the program text which are subject to interception (that is, for which *hooks* must be inserted) are identified through the definition of class and operation selectors. The former is a predicate used to select classes and the latter is used to select operation occurrences that will be hooked. These definable predicates allow for advanced algorithmic cut to be specified. Hooks are grouped in hooksets, which are given an identifier. Hooksets can gather hooks scattered in separate entities, and can be composed. Referring to AspectJ's vocabulary, a hook is a join point shadow [34], that is, the static correspondent of a dynamic join point. A hookset therefore corresponds to a pointcut shadow [47].

In AspectJ, a dynamic join point is determined by its join point shadow and so-called *residues* [34]. Residues are dynamically-evaluated conditions that determine whether a runtime occurrence of a join point shadow actually is a join point. In our model, support for dynamic cut is *explicitly* provided through activation conditions that can be attached to links. AspectJ supports only three kinds of residues, related to `if`, `instanceof`, and `cflow` conditions, as a result of the limited expressiveness of AspectJ's pointcut language. Our model naturally supports these residues, and more elaborated dynamic conditions can also be devised (for instance depending on some application-specific characteristic of the instance in which the hook is being executed), as in Josh.

	s−link	b−link
load	s−MO	hook
run	−	b−MO

Figure 7: Types of links and their application according to the time.

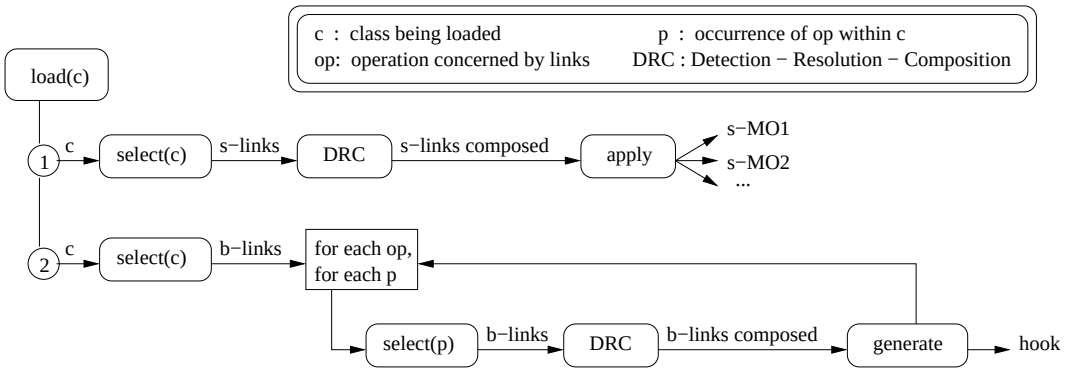

Figure 8: Reflex operates in two phases: the first phase applies s-links, the second phase installs b-links.

Moreover, in Reflex, an activation condition is reified: it is an object, implementing an interface, that can be manipulated at runtime. This is a necessary feature for supporting dynamic AOP systems.

As discussed in section 3.2, there are many ways to design a dynamic cut language depending on the desired expressiveness, for instance to express control flow. In this regard, we have identified *event collectors* as a useful building block: event collectors gather execution events to expose parts of a program execution (nesting, sequences, etc.), under any structure (counter, stack, tree, DAGs, graphs, etc.), for dynamic introspection, in particular to determine dynamic cuts. Event collectors are integrated uniformly in our model, since they are indeed just metaobjects. They differ from metaobjects implementing aspect behavior in the sense that they only record and expose information about the execution. Other entities, in particular activation conditions, can query them as required. For instance, exposing a simple control flow depth counter is done by a thread-local metaobject encapsulating an integer with a public accessor (Fig. 9). A more elaborate system to analyze execution history, such as the Java PathExplorer system [33], is indeed an advanced event collector.

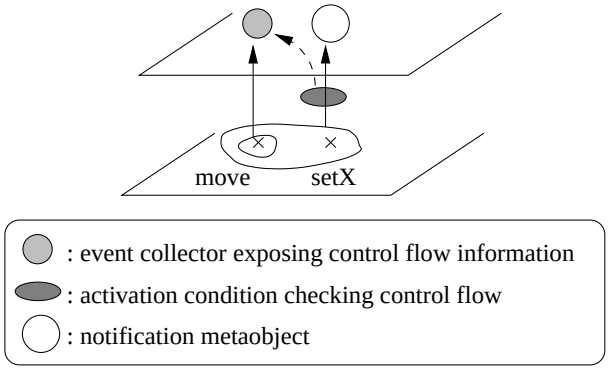

Figure 9: Example of an event collector providing control flow information to a dynamic cut.
The dynamic cut ensures that the notification metaobject only acts for top-level calls to `move` or `setX`.

Binding specification in the hookset model is separated from static cut (hooksets) and action (metaobject) definitions. A *link* is an object that specifies the binding between hooksets and metaobjects. It is characterized by several attributes, one of them being the activation condition that is used to specify the dynamic part of the cut.

The information bridge between the cut and the action, the *aspect protocol*, is realized in our model by the protocol between hooks and metaobjects, which is called the metaobject protocol (MOP). Reflex provides open MOP support, meaning that this protocol is not standardized for the whole application. Various MOPs can be defined, through *MOP descriptors*. A MOP descriptor specifies how the delegation from a hook to a metaobject is done: the type of the metaobject, the method to invoke, and the parameters to pass. This specification is done using Javassist facilities, which have been proven to be both expressive and efficient [14]. Reflex gives the possibility to fix a MOP description by default for a given operation, as well as to provide link-specific MOP descriptors. Hence, it is possible to be as specific as AspectJ's selective pointcut parameter exposure. Josh relies on the same Javassist abilities for custom context exposure [13]. The only difference between Reflex and Josh in this regard is that Josh *inlines* aspect code (parameters are made available on the stack), while Reflex *delegates* to aspect code (parameters are passed when calling the metaobject method). In other words, aspects are not reified in Josh. In Reflex, metaobjects reify aspect actions, which are then subject to other aspect cuts.

4.3 Structure

Reflex uses Javassist to provide behavioral reflection (see for instance [12], which shows how a simple behavioral reflective system can be built with Javassist). To extend Reflex with structural abilities, giving access to the Javassist API from within Reflex to structural metaobjects may seem sufficient. However, as argued in section 3.4, Reflex must provide a collaboration protocol to control the visibility of structural modifications applied by aspects. Such a collaboration protocol is not directly supported by Javassist.

Hence, Reflex has been augmented with a new object model, wrapping that of Javassist, in order to provide a collaboration protocol. In other words, Reflex structural API mimics that of Javassist for both introspection and intercession, but in addition, it supports both implicit and explicit collaboration between transformations. By default, all structural changes made to a class object through Reflex structural API will be invisible to others when they introspect the class. If it is known that some changes should be visible to some aspects, then an extra parameter (a flag) can be specified. When introspecting a class, an aspect can see the changes made by others by specifying the corresponding flags. A special flag is provided to view all changes made to a class. Newly created classes are subject to this same collaboration protocol.

Structural aspects are specified by defining a *structural link* (s-link). An s-link associates a class set and a structural metaobject, which transforms classes using Reflex structural API. The class set is defined by a class selector, which is a predicate that allows for complex algorithmic cut: the full-fledged class introspection API makes it possible to base the selection of a class on any criterion, such as manipulated types, messages sent, etc.

In order to make it possible for behavioral changes to be based on or refer to previously made structural changes, Reflex operates in a two-phase process, where structural aspects are applied first (recall Fig. 8).

4.4 Composition

The way composition issues are solved is a key characteristic of an AOP kernel. In this section we explain how Reflex deals with detection of aspect interactions, composition of aspect behaviors, and collaboration among aspects. Our proposal is inspired by the work of Douence *et al.* on a generic framework for the formal definition and interaction analysis of aspects [20, 21, 22, 23]. The authors are concerned by the formal description and analysis of aspect interactions, while we address the actual detection of interactions directly at the implementation level. Concerning aspect composition, the authors introduce several expressive composition operators, which we extend in order to take into account implementation concerns such as staging.

4.4.1 Detection of aspect interactions

In Reflex, interactions among behavioral aspects are detected in phase 2 when selecting the set of links applying at a given program point (recall Fig. 8). If this set includes more than one link, there is an interaction. If a composition strategy was specified, it is applied (this is explained later). If not, Reflex issues a warning and arbitrarily composes the interacting links. It is also possible to run Reflex in a mode that makes it stop whenever an interaction that is not explicitly resolved is detected.

The detection and notification of structural interactions are handled in a similar way. In phase 1, whenever a class belongs to more than one class set, there is an interaction (Fig. 8). Similar notification strategies are used. In the current version of Reflex, only these interactions are detected. Detection of more complex structural interactions, such as overriding in an aspect a method added in a superclass by another aspect, is to be further

studied. Structural conflicts (e.g., two aspects adding a method with the same name), however, are reported by the underlying transformation tool, Javassist.

4.4.2 Composing aspects

Douence *et al.* introduce several composition operators to resolve aspect interactions [20, 21]. Composition operators either relate to ordering aspects, such as $seq(A_1, A_2)$ which sequentializes aspects, or to ignoring aspects, such as $fst(A_1, A_2)$ (resp. $snd(A_1, A_2)$), which ignores A_2 (resp. A_1). Another kind of composition operators makes it possible to condition the application of aspects to the application of others. For instance, $cond(A_1, A_2)$ specifies that, at interaction points between A_1 and A_2, A_2 should only be applied if A_1 applies. This support for the explicit detection and manipulation of *aspect application* greatly contributes to the expressiveness of this approach to aspect composition. Furthermore, the set of composition operators is extensible: arbitrary, n-ary, operators can be considered (e.g., an if-then-else operator).

An implementation of this approach to aspect composition was proposed in a prototype for EAOP [24]. In that proposal, the expression representing aspect composition (nodes are composition operators, leaves are aspects) exists and can be manipulated at runtime. Boolean flags, attached to the leaves, make it possible to test whether an aspect applies or not. A drawback of this all-dynamic approach is its cost: each time an execution point belonging to a cut is reached, the composition expression has to be interpreted.

In the perspective of an efficient, stage-based, AOP kernel, we are interested in preserving such expressiveness (extensible set of composition operators, notion of aspect application), without necessarily that level of dynamicity. Hence, in our approach, a composition expression only exists at load time to drive the generation process: nodes are still composition operators, while leaves are now links. We stage aspect composition by statically fixing into the generated hook code what is known at generation time, and leave the rest for runtime evaluation. In cases where dynamic manipulation of aspect composition is needed, an EAOP-like metaobject composition framework can be plugged on top of Reflex[2].

Among the composition operators mentioned at the beginning of this section, those related to ignoring and ordering aspects do not depend on aspect application, and hence can always be resolved at weaving time. Conversely, operators related to conditional application do depend on aspect application. In our proposal, aspect application is implemented as *link application*, of which we support three levels:

- static application: a link is considered to statically apply at a given point *iff* there is a hook for this link at this point (Fig. 10a);
- active application: a link is considered to actively apply at a given point *iff* it statically applies *and* it is active. A link is active either if it is an always-active link, or if its dynamic activation condition evaluates to true (Fig. 10b);
- dynamic application: a link is considered to dynamically apply at a given point *iff* it actively applies *and* the metaobject explicitly states that it applies. This explicit statement is done by exposing a variable (the metaobject has to implement an interface with a `boolean applied()` method) (Fig. 10c).

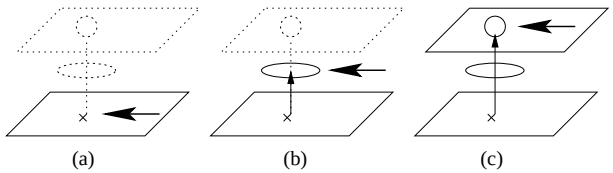

Figure 10: The three stages of link application: (a) static application, (b) active application, (c) dynamic application.
The dark arrow shows the level at which link application is determined.

Static application is determined automatically and statically. Active application is also determined automatically, but dynamically. Dynamic application is automatically checked at runtime, but it is the responsibility of metaobjects to explicitly state their application. An additional link attribute has been added to specify whether the application of a link should be determined statically, actively, or dynamically. Since our representation of

[2]This experiment has been carried out recently in the OBASCO research group (technical report in preparation).

links already includes the possibility of stating type restriction rules for metaobjects, we are able to guarantee that only a metaobject that implements the interface for checking dynamic application can be attached to a link whose application should be determined dynamically.

Once aspect interactions have been detected and composition has been specified, Reflex generates hook code that automatically checks for link application at the appropriate level, if needed. To sum up, the hook generation process is the following:

$$(1)\ select(p) \Rightarrow \{l_i\}_p$$
$$(2)\ pe(CE, \{l_i\}_p) \Rightarrow CE_p$$
$$(3)\ gen(CE_p, \{l_i\}_p, p) \Rightarrow hook$$

(1) For each program point p, the set of interacting links $\{l_i\}_p$ is determined by applying operation selectors (*select*); (2) the composition expression CE is partially evaluated (*pe*), considering $\{l_i\}_p$ as static input; a specialized composition expression for p, CE_p, is obtained; (3) hook code is generated (*gen*) from this specialized expression. This code manages the composition issue, in particular with automatic checks of link application.

For instance, let l_1 and l_2 be two links that interact at a given point p. Considering that the composition expression for p is $cond(l_1, l_2)$ (meaning l_2 should be applied only if l_1 does not apply), there are three possibilities:

1. l_1 has static application. We know l_1 applies, so we do not apply l_2. The generated hook code is simply: `{` $hook_1$ `; }`

2. l_1 has active application. We need to check in the hook code the application of l_1 through its activation condition:
   `{` $hook_1$ `; if(!`$active_1$`)` $hook_2$ `; }`

3. l_1 has dynamic application. We further need to check in the hook code the application of l_1 through the associated metaobject:
   `{` $hook_1$ `;`
   `if(!`$active_1$`|| !`$metaobject_1$`.applied())` $hook_2$ `; }`

The composition of structural aspects follows the same principles: a structural composition expression can be specified, and the same operators are provided. The only difference is that aspect application is determined statically: if a class belongs to the class set of a structural aspect, then this aspect is considered to apply to that class. Also, there is no mechanism for hook generation, since structural metaobjects are directly applied at load time.

4.4.3 Collaboration among aspects

In section 4.3, we already explained the collaboration protocol among structural aspects, which occurs during phase 1 (Fig. 8). This protocol is based on the idea of an explicit parameterization of change visibility, ensuring that by default, an aspect does not see changes made by others, while allowing some of the aspects to explicitly collaborate.

Concerning behavioral aspects, the issue is more complex. Behavioral changes induced by aspects are encapsulated in the class defining the behavioral metaobject, and hence are visible to another aspect only if the cut bound to the aspect affects the behavioral metaobject class. However, hook insertion actually represents a modification of base classes. First of all, Reflex generates several infrastructure members that are added to the class being transformed, used to reference and manipulate metaobjects and activation conditions. This infrastructure should not be considered part of the base application and exposed to further aspect-related transformation. Secondly, a hook itself is a piece of code typically inserted in a method body. Again, the code of a hook should not be exposed to further aspect-related transformation.

Reflex guarantees that both infrastructure members and hook code are invisible to aspects. [60] explains how hooks are inserted in base code by so-called *hook installers*. A hook installer typically knows how to hook a given operation (message send, cast, etc.). During phase 2, it is possible that a hook installer for a particular operation needs to perform some structural changes. For instance, for the *message receive* operation, a hook installer that adopts the technique of method wrappers will rename each method `m` using a hidden name, and add a new method named `m` to the class: this new method holds the code of the hook. Furthermore, a cache of wrapped method objects is usually added to the class, so that they can be passed to metaobjects at runtime, for introspection as well as for use as a continuation-like mechanism. After such an installer has performed

its transformations, the appearance of the class has changed: it has more methods than before, and an extra field. The following hook installer is likely to misbehave. Hence this transformation has to be hidden to the next installer. This is achieved by using specific code wrappers recording the transformations but returning the initial code in case the code is introspected by a new installer.

4.5 Open Support for Aspect Languages

In our approach, aspect behavior is implemented as metaobject classes, which are standard Java classes: the action language is Java, along with available APIs. Other entities, such as selection predicates and activation conditions are also implemented in standard Java classes. MOP descriptors are Java entities that use Javassist-specific text descriptions (see [14]). All other elements of an aspect language (Fig. 6) relate to configuring Reflex. Hence, a particular aspect language is interpreted in part as a *configuration language* for Reflex.

Reflex offers a configuration API for defining hooksets, links, composition, and so on, that is accessible statically and dynamically. For static configuration, the default way to proceed is by using configuration classes, which are classes with static `initReflex` methods that directly interact with the Core Reflex API [60].

We have opened the handling of command line arguments in order to support new configuration means. Each mean of configuration language is provided by a *configuration handler*: there is one configuration handler per aspect language L_i. New configuration handlers can be registered with Reflex by placing handler jar files in a specific directory (specified with an environment variable). A handler declares the command line flags it can handle, and Reflex passes the values for each flag to the appropriate handler. For instance, XML configuration is handled through a configuration handler that supports the `-configXML` flag, hence:

```
java reflex.Run -configClass MyConfig
                -configXML file.xml
                App
```

applies both a configuration class and an XML configuration file.

Reflex also registers which handler performed which configuration (e.g., link definitions). This is necessary to give control back to handlers in case of invalid definitions or aspect interactions, so that handlers can output messages to the programmer at the appropriate level of abstraction.

4.6 Base Language Compliance

Several link attributes have been introduced in Reflex in order to better support various semantic elements of Java.

4.6.1 Inheritance

Reflex provides a means to specify whether the cut of an aspect applies to subclasses or not through a dedicated link attribute. This attribute states whether a link is *private* or *public*. If a link is private, it does not apply to subclass instances, whereas if it is public, it is propagated to subclasses.

4.6.2 Concurrency

Reflex makes it possible to specify the initialization mode of the metaobject associated to a link: it can be *lazy-unsafe*, meaning the metaobject is created lazily, without synchronization code; if it is *lazy-safe*, lazy instantiation is ensured to be safe, but has a cost in synchronization; finally, it can be *eager*, in which case creation is done eagerly in a thread-safe manner, without requiring synchronization. Another link attribute specifies the thread visibility of a link: it can be *global* or *local*. A thread-local link will result in each thread using a dedicated metaobject instance, whereas a thread-global link specifies that all threads access the same metaobject instance.

4.6.3 Security

Reflex supports the specification of restrictions over the accepted types of a metaobject, and also makes it possible to enforce some link attributes. Integration of a security mechanism in the line of Vayssière's work [9, 11] is future work, but the compatibility of Reflex with such mechanisms is already guaranteed since metaobjects are runtime entities separated from the base application, and only trusted code generated by Reflex is inserted into application classes.

4.6.4 Integration with environment

Reflex supports two main operational modes: as a post-processor, it can statically transform sets of classes and jar files, generating modified classes; as a class loader, it transforms classes on-the-fly as they are loaded. When operating at load time, Reflex can leave the modified classes on disk so that they do not need to be transformed again at the next execution. Also, at runtime, new classes can be generated, and the configuration API is still accessible. Of course, being a portable Java tool based on bytecode transformation imposes a major implementation restriction: code transformation can be done statically or at load time, not on already-loaded classes. There has been some experiments to remove this limitation of bytecode-based approaches [15], but they still need to be further explored. Another issue arises with bytecode-based approaches, when aspects are expressed in terms of high-level constructs that disappear during compilation to bytecode. This means that considering source-level transformations or open compilation could also make sense.

4.7 Interactions application/aspects

Reflex provides continuation-like mechanisms through the standard Java reflection API, so that metaobjects controlling a link with replace control can resume an intercepted operation. Furthermore, the reflection API of Java makes it possible to interact with the base application in many ways, for instance by changing field values in objects in a generic manner.

Concerning interactions from the base application to aspects, Reflex offers a full-fledged runtime API for metaprogramming (see [60] for details). This API makes it possible to manipulate metaobjects associated to links, at the appropriate scope (object, class, hookset). Changing metaobjects is subject to safety restrictions: a link can declare a set of type restrictions that metaobjects should comply with, or can even declare that associated metaobjects cannot be changed at all. Due to the implementation approach of Reflex, we cannot fully reify a cut: its static part (hooksets) is fixed at hook generation time. However, activation conditions –used in particular for implementing dynamic cuts– are reified, and hence fully accessible through the API.

5 The SOM case study

SOM (Sequential Object Monitors) [10] is a scheduler-like approach to concurrency in Java. SOM makes it possible to separate the synchronization concern from the base application.

5.1 SOM with Reflex

The standard implementation of SOM is based on Reflex and served as our first realistic case study of the AOP kernel approach we are exposing. In SOM, base objects that should be turned into sequential monitors have their method invocations controlled by a monitor metaobject. This metaobject reifies these invocations as requests that are then scheduled by a custom scheduler object (Fig. 11).

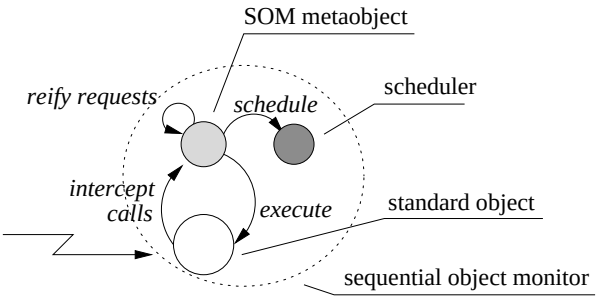

Figure 11: A sequential object monitor.

5.1.1 The SOM configuration language

SOM programmers surely do not want to directly configure Reflex to use SOM. In fact, the basic configuration needed for SOM simply consists in specifying which class should be scheduled by which scheduler. Hence,

programmers should not bother with link definitions, metaobject instantiation and so on. Hence we designed a lightweight configuration language for SOM, that simply consists of declarations of the form:

```
schedule: aClass with: aScheduler
```

Such a declaration specifies that instances of `aClass` should be sequential monitors scheduled by an instance of `aScheduler`. The SOM configuration handler we developed translates this into the equivalent Reflex configuration, for instance (simplified):

```
ClassSelector cs = new ClassNameSelector("aClass");
OperationSelector os = new PublicMethodsSelector();
Hookset hs = new Hookset("SOM", msgReceive, cs, os);
Link l = new Link(hs, new MODefinition.Class(SomMO,
                                    "aScheduler");
l.setScope(Scope.OBJECT);
l.setControl(Control.BEFORE_AFTER);
l.setInitialization(Initialization.EAGER);
l.setThreadVisibility(ThreadVisibility.GLOBAL);
defineLink(l);
```

A link in SOM is per object, acts before and after, is initialized eagerly and is thread-global. Registering the SOM configuration handler makes it possible to apply SOM (statically or at load time) using Reflex with the `som` flag:

```
java reflex.Run -som som.conf App
```

5.1.2 The SOM MetaObject Protocol

From an efficiency viewpoint, SOM has the advantage of avoiding useless thread context switches. For this advantage not to be completely annihilated by our implementation approach, we used the MOP specialization features of Reflex to define a specific, and efficient, MOP for SOM. Defining this MOP meant:

- providing a tailored interface for SOM metaobjects. This interface consists of an `entry` and an `exit` method (SOM metaobjects apply both before and after a method execution).
- providing a thread-safe and efficient implementation of SOM metaobjects. A SOM metaobject is a synchronized object that handles request queues and delegates to the scheduler the task of selecting the ones to serve.
- providing a MOP descriptor for SOM links. This descriptor specifies that the SOM metaobject interface should be used for the message receive operation. This descriptor allows for efficiency by ensuring that only required information is reified and passed to metaobjects: the `entry` method only requires as parameter the name of the called method and its parameters, while the `exit` method does not take any parameter.

Benchmarks validating the efficiency and scalability of the resulting implementation can be found in [10].

5.1.3 The SOM runtime API

A complementary runtime API has also been designed to wrap some of Reflex functionalities which are useful in the case of SOM. Most importantly, this API makes it possible to explicitly create a single instance that behaves as a monitor:

```
Vector v = (Vector) SOM.newMonitor(Vector.class,
                                   aScheduler);
```

This uses Reflex ability to dynamically create an implicit reflective subclass (with the classical restrictions associated to this approach). The SOM API can also be used to dynamically change the scheduler associated to a monitor.

Basing our implementation of SOM on the Reflex AOP kernel was a positive indicator for our approach: it was quick to implement, compared to the range of features that could directly benefit to SOM, and exposed satisfying efficiency and scalability.

5.2 Compatibility with other approaches

Providing SOM on top of the Reflex AOP kernel ensures compatibility with other approaches based on this same kernel. Consider for instance a profiling aspect that times method execution and logs the result, possibly implemented with a general-purpose aspect language:

```
aspect Timer {
  Stopwatch watch = new Stopwatch();
  //pointcut timed() defined
  before(): timed() { watch.start(); }
  after(): timed() { watch.stop(); watch.log(); }
}
```

This aspect will be implemented in Reflex with a before-after link, and a timer metaobject managing the stopwatch object. If the timing aspect interacts with SOM, Reflex will generate a warning saying that an aspect interaction between `"SOM"` and `"Timer"` was detected, and that resolution was unspecified. Leaving resolution unspecified, Reflex adopts an arbitrary solution, sequentialization.

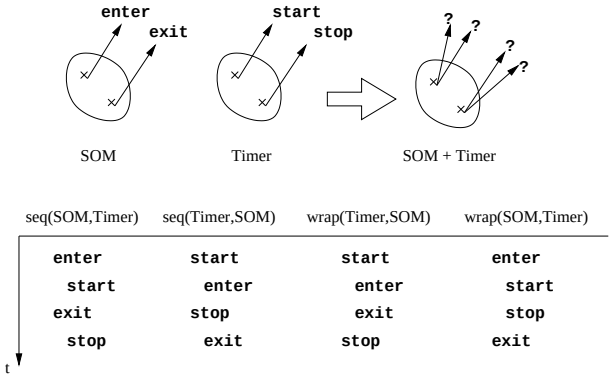

Figure 12: Interaction between SOM and a timing aspect.
The two aspects act before and after a method: SOM calls `enter` and `exit`, while Timer calls `start` and `stop`. There are 4 ways of resolving this interaction in order for both aspects to apply: only the two last are indeed meaningful in this case.

Default sequentialization as provided by the *seq* operator does not yield any meaningful result (Fig. 12). Another sequentialization operator is particularly adapted to compose before and after links, *wrap*. Using this operator, two interesting semantics can be specified: *a)* $wrap(SOM, Timer)$ will result in the timer measuring only method execution (hence not measuring SOM's cost at all), and *b)* $wrap(Timer, SOM)$ will result in the timer measuring the whole execution time of a method, including scheduling time and possible suspension time.

6 Related Work

Historically and technically, reflection is at the heart of aspect-oriented programming, which can be seen as a disciplined, principled, way of doing metaprogramming. Kojarski *et al.* explore in [43] the relation and interactions between reflection and AOP. This discussion however only considers *generic* AOP, in the line of AspectJ and AspectS [35]. They show and discuss the tradeoffs in implementing AOP with reflection and, reciprocally, to implement reflection via AOP. A result of this study is that a MOP should be rich enough to enable AOP, while AOP makes it possible to apply reflection more selectively. Selective application of reflection is also the object of *partial reflection* [37, 60].

Several tools for domain-specific software engineering have been proposed in the literature (e.g., [4, 29, 57]). They share our motivation with respect to the advantages of supporting multiple domain-specific languages, but do not directly deal with aspect-oriented programming. Several approaches directly relate to our work: XAspects [55], Aspect-Oriented Logic MetaProgramming (AOLMP) [8], the Concern Manipulation Environment (CME) [32, 36], and Josh [13].

XAspects is a plug-in mechanism for domain-specific aspect languages [55]. In XAspects, each domain-specific solution communicates with a common language, which is the AspectJ language. The authors motivate their choice of AspectJ by the possibility of better expressing crosscutting abstractions, conversely to what could be done with Intentional Programming [56] or macro systems such as Maya [2], for instance.

We definitely share the viewpoint that an AOP kernel should intrinsically support crosscutting, but we believe that AspectJ is not the best candidate for an AOP kernel. AspectJ is a valuable and mature proposal of a generic aspect language: its practitioner perspective entails that too many design choices are fixed, which results in limited versatility. For instance, composition of aspects is limited to static precedence, which is insufficient in general. Another example of the non-versatility of AspectJ is the fact that complex algorithmic cuts cannot be specified. Also, AspectJ does not fully support runtime manipulation and evolution of aspects. Additional code has to be developed each time such features are needed (see for instance [18]).

The work on XAspects clearly identifies and separates structural and behavioral transformations. XAspects proposes a 6-phase compilation process. Similarly to our proposal, there is a phase for structural changes and a phase for behavioral changes. However, it is unclear whether their crosscutting analysis is applicable to a scenario with dynamic class loading. Our approach is also more uniform in the sense that it does not simply provide plain bytecode to transformers, but rather uses a high-level API for load-time structural introspection and intercession among all phases, based on Javassist. XAspects only supports passive collaboration between plugins, whereas Reflex offers a more flexible collaboration protocol allowing both implicit and explicit collaboration. Concerning the phase dealing with generation of semantics (i.e., behavioral changes), XAspects imposes the constraint that a domain-specific aspect can only be turned into one AspectJ aspect. Conversely, in our approach, the number of links and metaobjects involved in the implementation of one domain-specific aspect is not limited. Finally, being based on AspectJ, XAspects cannot provide a clear and expressive composition semantics of domain-specific aspects.

In [8], an approach to building composable aspect-specific languages with logic metaprogramming is proposed. Aspect-specific languages are uniformly defined and composed using logic programming. This approach is very expressive, in particular with respect to composition, but exposes programmers to a high-level of complexity: aspect languages by themselves do not really shield the programmer from the inherent power of the logic metaprogramming approach. This comes from the fact that aspects are defined in the same logic framework as the languages. Conversely, an approach where aspect definitions are separated from aspect language definitions provides a simpler and more protective environment for programmers.

The Concern Manipulation Environment (CME) developed at IBM [36] 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, there are fundamental differences. First of all, the CME targets both asymmetric and symmetric approaches to separation of concerns [31], while the present work only addresses asymmetric approaches. To be more precise, following the classification of symmetries presented in [31], our proposal exhibits element asymmetry with support for aspect-aspect composition, join-point asymmetry, and partial relationship symmetry (asymmetric placement, but symmetric scope with respect to aspect). Also, the Concern Assembly Toolkit (CAT) [32], part of the CME, aims at many different representations of software, such as Java source code, Java class files, and other representations, like UML diagrams. Though restricted to object-orientation, the great variety of target representations for CAT has a serious impact on the concern assembly language (the equivalent of the kernel language in our approach): assembly directives are usually specified open-endedly as strings. This has to be contrasted with our higher-level conceptual model used to reason about behavioral, structural and compositional issues. Finally, composition is only considered at the level of composing class hierarchies [58] or at the level of method bodies through method combination graphs. Detection and resolution of aspect interactions as treated in this report is not considered.

Finally, Josh [13] is a proposal very close to ours, both in its motivation and its realization. Josh is an open AspectJ-like language, which makes it possible to experiment with new cut languages and means of describing advices. Josh is based on Javassist [12, 14], like Reflex, and hence has similar expressive capabilities. Nevertheless, Josh weaves aspects by inlining advice bodies, whereas Reflex keeps aspect behavior in separate objects. Recall that keeping aspect behavior in separate objects is necessary when considering Java security, aspects of aspects, and per object aspects (not supported by Josh). Finally, Josh lacks support for aspect composition. Conversely, Reflex provides automatic detection of aspect interactions and expressive means for explicitly resolving these interactions.

7 Conclusion and Perspectives

G. Kiczales qualified AOP as a *principled subset of reflection* [39]. But indeed, this subset is not fixed, since it depends on the considered aspects and degree of specificity of aspect languages. We have argued for the fact that an appropriate reflective model supporting both structural and behavioral alterations of programs is generic enough to handle a wide range of variabilities among aspect-oriented approaches. Our model is based on the hookset model for the behavioral part and on the class-object model for the structural one. This generic reflective layer can be used to automatically detect and explicitly resolve interactions among aspects. The presence of the reflective layer also ensures that several AOP approaches can cooperate. Domain-specific AOP can then be achieved by a language layer on top the reflective one. The reflective layer provides the transformational and compositional power, while the language layer guides programmers by providing appropriate barriers and syntactic support.

Future work on the Reflex AOP kernel includes a deeper study of the means to report about the detection of aspect interactions at the appropriate conceptual level (the one in which the aspect was formulated), and of existing tools and approaches to facilitate the construction of domain-specific translators. Also, several aspect languages will be implemented on top of this kernel. In particular we plan to develop a simple and general-purpose configuration language for Reflex, as a teaching support for reflective and aspect-oriented programming.

Finally, an interesting perspective is to study the tight integration of AOP kernels within languages, providing full runtime support. With respect to Java, a possible direction to study a VM-integrated AOP kernel for Java is to experiment with evolutions of Steamloom [7], a research virtual machine for Java that supports dynamic AOP. Current work on Steamloom focuses on dynamic weaving and performance issues. It could be extended to take into account the issues presented in this report. It would also be valuable to study an AOP kernel for a highly-dynamic language, such as Smalltalk. This would be an environment of choice to experiment with dynamicity in AOP and therefore widen our analysis of AOP kernels for object-oriented languages. We believe that AOP kernels are a good way to reach some level of standardization in AOP without restricting the range of possibilities for new proposals.

8 Acknowledgements

We would like to thank Mario Südholt, Leonardo Rodríguez and Guillaume Pothier for their valuable comments on a draft of this report.

This work was partially funded by Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

References

[1] M. Ancona, W. Cazzola, and E. B. Fernandez. Reflective authorization systems: possibilities, benefits, and drawbacks. In *Secure Internet programming: security issues for mobile and distributed objects*, pages 35–49. Springer-Verlag, 1999.

[2] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–281. ACM Press, 2002.

[3] 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.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the Fifth Internation Conference on Software Reuse*, pages 143–153. IEEE, 1998.

[5] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, Oct. 2001.

[6] L. Bergmans, M. Aksit, and B. Tekinerdogan. Aspect composition using composition filters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, pages 357–382. Kluwer Academic Publishers, 2001.

[7] C. Bockish, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In Lieberherr [44].

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

[9] D. Caromel, F. Huet, and J. Vayssière. A simple security-aware MOP for Java. In Yonezawa and Matsuoka [65], pages 118–125.

[10] D. Caromel, L. Mateu, and E. Tanter. Sequential object monitors. In M. Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.

[11] D. Caromel and J. Vayssière. Reflections on MOPs, components, and Java security. In Knudsen [42], pages 256–274.

[12] 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.

[13] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In Lieberherr [44], pages 102–111.

[14] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376. Springer-Verlag, 2003.

[15] S. Chiba, Y. Sato, and M. Tatsubori. Using HotSwap for implementing dynamic AOP systems. In *Proceedings of the ECOOP Workshop on Advancing the State-of-the-Art in Runtime Inspection*, July 2003.

[16] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, 2000.

[17] M. Dahm. Byte code engineering with the BCEL API. *Technical Report B-17-98, Berlin*, 2001.

[18] P.-C. David, T. Ledoux, and N. M. Bouraqadi-Saâdani. Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[19] B. De Win, B. Vanhaute, and B. De Decker. Security through aspect-oriented programming. In B. D. Decker, F. Piessens, J. Smits, and E. V. Herreweghen, editors, *Advances in Network and Distributed Systems Security*, pages 125–138. Kluwer Academic Publishers, 2001.

[20] 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.

[21] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [44], pages 141–150.

[22] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. To appear.

[23] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [65], pages 170–186.

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

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

[26] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *Workshop on Foundations Of Aspect-Oriented Languages (FOOL) at AOSD 2002*, Twente, Netherlands, Apr. 2002.

[27] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[28] L. Gong. *Inside Java 2 Platform Security: Architecture, API design, and implementation*. Addison-Wesley, 1999.

[29] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, Oct. 2001.

[30] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, Washington, D.C., USA, Sept. 1993. ACM Press. ACM SIGPLAN Notices, 28(10).

[31] W. H. Harrison, H. L. Ossher, and P. L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research, 2002.

[32] W. H. Harrison, H. L. Ossher, P. L. Tarr, V. Kruskal, and F. Tip. CAT: A toolkit for assembling concerns. Technical Report RC22686, IBM Research, 2002.

[33] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 (No. 2) of *Electronic Notes in Theoretical Computer Science*, Paris, France, July 2001. Elsevier Science.

[34] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In Lieberherr [44], pages 26–35.

[35] R. Hirschfeld. AspectS – aspect-oriented programming in Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

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

[37] M. H. Ibrahim. Report of the workshop on reflection and metalevel architectures in object-oriented programming. In *OOPSLA/ECOOP'90*, Ottawa, Canada, Oct. 1990.

[38] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, volume 1343 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[39] G. Kiczales. The future of reflection. Invited talk at the Third International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001), Sept. 2001.

[40] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In Knudsen [42], pages 327–353.

[41] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finland, June 1997. Springer-Verlag.

[42] J. Knudsen, editor. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, Budapest, Hungary, June 2001. Springer-Verlag.

[43] S. Kojarski, K. Lieberherr, D. H. Lorenz, and R. Hirschfeld. Aspectual reflection. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Boston, Massachusetts, Mar.18 2003. AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies.

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

[45] C. V. Lopez and G. Kiczales. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, 1997.

[46] B. Magnusson, editor. *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in Lecture Notes in Computer Science, Málaga, Spain, June 2002. Springer-Verlag.

[47] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *AOSD Workshop on Foundations of Aspect-Oriented Languages*, pages 17–26, 2002.

[48] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, Feb. 1997.

[49] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [44].

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

[51] D. Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[52] A. Rashid. A hybrid approach to separation of concerns: The story of SADES. In Yonezawa and Matsuoka [65], pages 231–249.

[53] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behavior. In Magnusson [46], pages 205–230.

[54] D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 30–39, Boston, Massachusetts, Mar. 17-21 2003. AOSD 2003, ACM Press.

[55] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain-specific aspect languages. In *OOPSLA 2003 Domain-Driven Development Track*, October 2003.

[56] C. Simonyi. The death of computer languages. Technical Report TR-95-52, Microsoft Research, 1995.

[57] Y. Smaragdakis and D. Batory. DiSTiL: A transformation library for data structures. In *Domain-Specific Languages (DSL) Conference*, pages 257–270, 1997.

[58] G. Snelting and G. Tip. Semantics-based composition of class hierarchies. In Magnusson [46], pages 562–584.

[59] E. Tanter, N. Bouraqadi, and J. Noyé. Reflex – towards an open reflective extension of Java. In Yonezawa and Matsuoka [65], pages 25–43.

[60] E. 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).

[61] E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In Batory et al. [3], pages 283–298.

[62] M. Tatsubori. *A Class-Object Model for Program Transformations.* PhD thesis, Graduate School of Engineering, University of Tsukuba, Japan, Jan. 2002.

[63] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. 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 2–21, Saint-Malo, France, 1999. Springer-Verlag.

[64] I. Welch and R. J. Stroud. Using reflection as a mechanism for enforcing security policies on compiled code. *Journal of Computer Security*, 10(4):399–432, 2002.

[65] 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*, Kyoto, Japan, Sept. 2001. Springer-Verlag.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399