Presented at the European Interactive Workshop on Aspects in Software 2004

Motivation and Requirements for a Versatile AOP Kernel

Éric Tanter[1,2] Jacques Noyé[3,2]

[1]University of Chile, DCC/CWR
Avenida Blanco Encalada 2120, Santiago, Chile

[2]OBASCO project, École des Mines de Nantes – INRIA
4, rue Alfred Kastler, Nantes, France

[3] INRIA
Campus Universitaire de Beaulieu, Rennes, France

{Eric.Tanter,Jacques.Noye}@emn.fr

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 paper, 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. An AOP kernel for Java is currently under development.

1. MOTIVATION

The variety of toolkits and proposals for Aspect-Oriented Programming (AOP) [27, 37] and related modularization technologies for separation of concerns (SOC) [47] 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 [13, 15, 19, 39] to experiment with the design space, but they require redeveloping an ad hoc software layer to bridge the gap with a proper high-level interface.

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 a versatile kernel for AOP that makes 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 [36] relies on the notions of join points, pointcuts, and advices; Event-based AOP (EAOP) [24, 26] uses concepts such as crosscuts, monitors, events, and aspects; models from the reflection community rather talk in terms of hooks and metaobjects [50, 54, 58], 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.* [41] 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.

Some approaches adopt general-purpose aspect languages [6, 36, 46], 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. Domain-specific approaches present many benefits: declarative representation, simpler analysis and reasoning, domain-level error checking and optimizations [18]. Several aspect-specific languages were actually proposed in the "early" ages of AOP [35, 42, 44], as well as recently, for instance DJCutter [45] for distributed systems. Wand argued that AOP should refocus again on domain-specific languages [57].

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 [49, 53].

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 [49]. 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* [23]. 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. Indeed, the symptom of the incorrectness may be very hard to track down to its source.

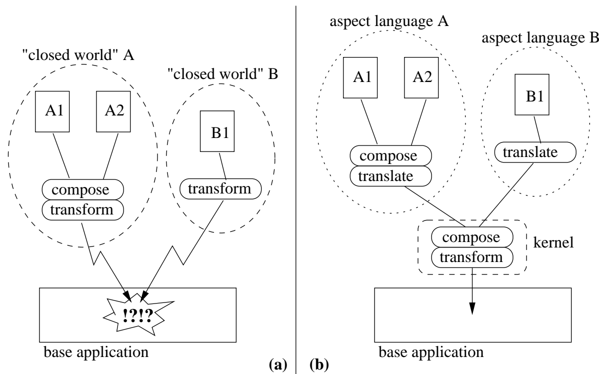

Figure 1: The compatibility issue between AOP approaches.
(a) Different AOP approaches, making a closed world assumption, are applied together: aspect interactions are blindly treated, jeopardizing the resulting semantics.
(b) A common AOP kernel is used as a mediator to detect and resolve interactions: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

1.3 A Versatile AOP Kernel

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, typically done from scratch with a closed world assumption. 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 ensured aspects are properly composed (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 a versatile AOP kernel 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 [33, 16], 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.

In the following section, we identify the different features of an AOP systems. From these features, Section 3 draws a list of requirements for a versatile AOP kernel.

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, which an AOP kernel takes care of. We will come back to this point in section 3.

Another feature is the *symmetry* of the approach, as discussed in [32]. In *asymmetric* approaches to separation of concerns, there is the notion of a base application to which aspects are applied, conversely to *symmetric* approaches to separation of concerns, like hyperspaces [46] or subject-oriented programming [31] where such a distinction is not done. Still, both kinds of approaches use some specific language to glue pieces together. We are here interested in asymmetric approaches: in such a 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 sometimes been identified as a key property of AOP [27]. It refers to the fact that the base application remains unaware of the aspects that are applied to it. However it has since been considerably softened[1]. First, base code needs to be structured in a sense that

[1]A discussion on the AOSD mailing list confirms this remark: the idea of "non-invasiveness" is now put forward.

makes it possible for aspects to intervene, implying some "awareness" of aspects. This was indeed already noticed in [37], and was a basic idea of the work on open implementations [48]. As Wand puts it, aspects reason about the *ontology* of a base program: this joint ontology is knowledge that is held in common between the base program and the aspects [57]. Second, industrial applications of AOP have clearly highlighted that explicitly *annotating* base code with semantic meta-information can be of great value, as an explicit "interface" exposed to aspects: in Wand's terminology, annotations can be seen as a way to define a joint ontology that is more abstract, domain-specific, than the general-purpose, language-based, join point model of typical AOP languages like AspectJ. Finally, we can add that, 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 [50]. The *binding time* refers to the time at which an aspect can be bound to a base program. 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 done and/or undone dynamically.

Finally, aspect languages can address two types of alterations: behavioral and structural ones. Most work on AOP has been around behavioral alterations. But an aspect language may include a part dedicated to structurally altering the base program, e.g., adding new members or interfaces to classes. This is known as *introductions* or *inter-type declarations*. Aspect languages like AspectJ [36] or Josh [14] include both behavioral and structural aspects. A difference is that AspectJ does not deal uniformly with inter-type declarations, while Josh does.

2.1 Main Features

An aspect language can be described according to several features. This section focuses on the most common ones: the cut, action and binding languages; and mechanisms for aspect parameterization, instantiation, and scope.

2.1.1 Cut language

The cut language is the language provided to specify the places where aspects affect the base application[2].

A *behavioral cut* denotes a set of *execution points* in a program. Such a *dynamic* behavioral cut can be projected (non-injectively) in the program text to a *static* behavioral cut, which denotes a set of *program points* corresponding to expressions in the program called the *shadows* [43] of the execution points.

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 expressions. 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[3].

Cut languages may include the possibility to refer to points in aspect programs (to apply aspects to aspects), may allow complex algorithmic cut to be specified [14], and may be tailored to a particular domain [45].

For instance, AspectJ cuts (pointcuts for behavior and type patterns for structure) are generic (i.e., not domain-specific) and can neither refer to aspect program nor describe algorithmic cuts. Specific features of behavioral cut languages include the expressiveness to refer to both program and execution points (for instance related to control flow). For instance, Soul/Aop as presented in [10] only affects program points while AspectJ pointcuts can refer to both program and execution points. But AspectJ pointcuts are limited with regards to control flow, compared to what has been proposed by others, for instance [25].

2.1.2 Action language

The action language is used to implement the aspect semantics. Both *structural actions* and *behavioral actions* may be supported. The action language may allow the definition of stateful aspects.

Behavioral actions 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.

A structural action language provides means to alter the data structures of a program, for instance by adding new members to a class or making a class implement an interface. In a language like Smalltalk [29], such alterations can be done dynamically, while in Java they can only be done statically.

2.1.3 Binding language

The binding language is used to specify which action should be bound to which cut. Many aspect languages do not decouple this language from one 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.

Interestingly, the four binding combinations are valid (see Fig. 2). The definitely most-used combination is to bind a behavioral action to a behavioral cut. This is the usual perception of *dynamic crosscutting*. However, considering a runtime environment that supports runtime structural modifications of classes, a structural action can very well be associated to a behavioral cut. Binding a structural action to a structural cut is the classical case of introdutions. Less frequent but yet worthwhile is the binding of a behavioral action to a structural cut: for instance, checking invariants or coding rules, like AspectJ's compile-time warnings and errors, can be seen as behavioral actions associated to a structural cut. To sum up, the different possible bindings between structural and behavioral cut and action are shown in Fig. 2, with their typical usage.

In the case of behavioral actions, the binding language usually also makes it possible to specify the kind of control the aspect has over the considered execution points (before,

[2]Depending on the proposal, the cut language is either referred to as a *crosscut language* [10, 21] or as a *pointcut language* [14, 37].

[3]Our analysis of structural cut is actually restricted to the class level: we do not consider approaches where only some particular instances of a class are affected by an introduction.

	behavioral cut	structural cut
behavioral action	*dynamic crosscutting*	*invariants, AspectJ's warning/error*
structural action		*introductions a la AspectJ*

Figure 2: Summary of the possible bindings between structural and behavioral action and cut, and their typical usage.

after, instead of, etc.), although such a notion becomes irrelevant for structural cuts, simplifying the binding specification.

2.1.4 Aspect parameterization

Aspect parameterization refers to the possibility of passing context information from cuts to actions. Providing context information enhances the expressiveness of the action language, and allows for more reusable aspects through genericity. However, for behavioral cuts, this has a cost at runtime, especially if the context information is passed in a generic manner. AspectJ addresses this issue with selective parameter exposition in pointcuts, similarly to the context exposure mechanism of Josh. For structural cuts, the issue of aspect parameterization can be simplified: the information needed is usually reduced to the considered data structure (in a class-based language, the class subject to modification).

2.1.5 Aspect instantiation and scope

Aspect instantiation and scope are features of the aspect language that specify how aspects are instantiated and what their scope is. For actions bound to behavioral cuts, 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. Conversely, an action bound to a structural cut is usually applied statically, hence a single global instance suffices.

2.2 Composition of Aspects

Few aspect languages explicitly support aspect composition. Josh does not provide composition support at all, while AspectJ provides a limited aspect composition language that can only state precedence between aspects. More expressive approaches to composition have been justified [10, 21, 22]. These proposals focus on behavioral aspects. Composition of structural actions has been poorly addressed in the AOP community, but still, very related work exists in the language community: work on metaclass composition [8], mixins [9], traits [51], etc. Finally some approaches aim at automatic *resolution* of conflits [17], while others argue for automatic *detection* and explicit resolution [21, 51].

2.3 Interactions Application/Aspects

Interactions between an application and applied aspects 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.

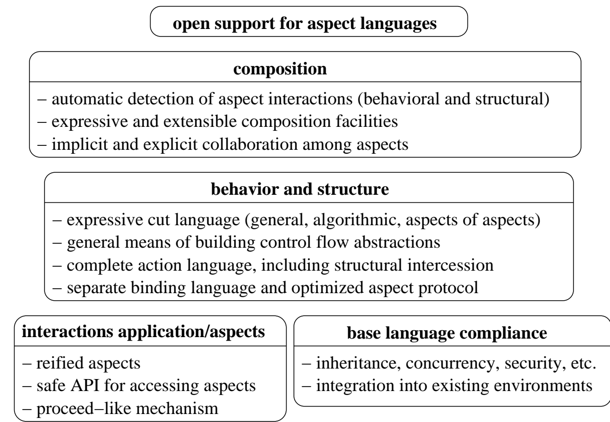

Figure 3: Summary of identified requirements for a versatile AOP kernel.

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 an AOP kernel 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* and is therefore interesting for supporting *cut* languages, while intercession deals with program *transformation*, and hence supports *action* languages. 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,

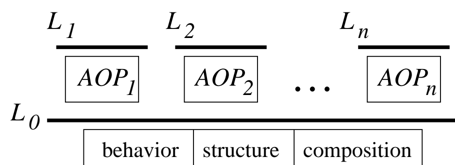

Figure 4: Elements of the AOP kernel approach. Each AOP system offering an aspect language L_i is implemented as a translator AOP_i to the common kernel language L_0.
The kernel language has 3 main parts: one for behavioral manipulation of the base application, one for its structural manipulation, and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

but rather as a *translator* from L_i to L_0 (AOP_i).

The behavioral and structural parts of L_0 must be an adequate target 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 remaining of this section.

3.2 Behavior and Structure

3.2.1 Cut

The kernel language must support expressive cuts, supporting aspects of aspects and complex algorithmic cuts. With regards to behavioral cut, behavioral notions highlighted by research in aspect-oriented programming (e.g., control flow) ought to be supported by the conceptual model at a generic level in order to allow various notions to be implemented in various ways. Generic introspective abilities of both program text and program behavior pave the way for such powerful cut support. For class-based languages, the intuitive representation of program text is based on the class-object model [56], 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 [29]. Tools like Javassist have extended this model down to the reification of expressions within method bodies [15].

The necessity of introspecting program behavior implies being able to discriminate amongst execution points that result from the same program point (expression). This discrimination can be done by filtering execution points based on their runtime description, and by looking at control flow relations. As a matter of fact, different variants of control flow can be designed. For instance, 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 [21, 22, 23, 24, 52, 28] has justified the benefits of being able to define aspects that apply depending on the execution history. Douence *et al.* have studied the formalization of such aspects [21, 22, 23, 24]. Sereni *et al.* have proposed control flow as regular expressions on the call stack [52]. Filman *et al.* have further extended the design of a language of events in order to fully express relationships among events, such as timeframe of occurrence [28]. Recently, Douence and Teboul have proposed an expressive cut language for control flow [25]. Thus, a generic means to build control flow abstractions is required.

3.2.2 Action

To handle all possible action languages, the kernel action language should support stateful aspects, so that they can maintain information across execution. Since this language should also be complete, using the base language is the logical choice. To some extent, a behavioral action can be merged within base code statically, like in Josh. Furthermore, for structural actions, a complete set of structural transformations has to be provided. This part consists in offering full structural intercession on the application. Since the base language may not directly support this (like Java), the use of a structural reflection tool (like Javassist) may be required.

3.2.3 Binding

The binding language deserves special attention, in particular when considering behavioral cut. 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 reflection and runtime metaobject protocols, the position is the opposite: the binding language is tied to the cut language. The 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 more 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 the *aspect protocol*.

3.3 Composition

Two aspects are said to *interact* if they affect the same program (or execution) point [23]. It is true that aspects may interact semantically without affecting the same points. However, it is very hard to capture this kind of "abstract" interactions between aspects. We therefore only focus on the former kind of interactions. In this respect, we concur with Douence *et al.* that the *resolution* of aspect interactions

ultimately depends on the application semantics and hence cannot be decided automatically, although such interactions can be *detected* automatically [21, 38]. 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 [10, 21, 22]: composing aspects does not solely refer to specifying the order in which they apply, but also, for instance, to possibly condition their application to the presence and application of other aspects. Composition issues relate to both structure and behavior. With regards to aspect precedence as supported by AspectJ, it is in fact a *nesting* of aspects: for aspects that act around a given operation, a `proceed` mechanism is provided in order to dynamically build the appropriate flow of control between nested advices and the base level.

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 made to base entities among aspects. If behavioral actions are reified and aspects of aspects are supported, the collaboration between behavioral changes is explicitly handled in aspect definitions: an aspect does not see another aspect unless its cut affects it.

3.4 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 paper. A discussion of the impact of distribution over the design of a cut language can be found in [45]. Finally, the issue of the implementation approach and integration of the kernel into existing environments is discussed.

3.4.1 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, as well as offering the possibility to stop downward propagation from a certain class.

3.4.2 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.

3.4.3 Security

Java features a security model based on a "sandbox" customizable with policies [30]. 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 [59, 2, 20]. 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 [11, 12]. 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 [30]. 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.

3.4.4 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 an AOP kernel in various settings, since it allows simpler and more widespread experiments to be carried out. If such an AOP kernel turns 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 the sake of efficiency, a kernel should be able to use *staging*, fixing some concerns statically in order to enhance performance, in case a particular approach does not require dynamicity. For instance, regarding aspect composition, it can be resolved statically [10, 36] or dynamically [21, 54]. An AOP kernel may 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 an AOP kernel can thus be seen as a tradeoff between choices fixed at weaving time and others left open at runtime. Also, an AOP kernel should possibly be used offline (e.g., as a post-processor), or online (e.g., as a special class loader).

3.5 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. As discussed in Section 3.3 a `proceed`-like mechanism is necessary in order to support approaches that allow aspects to wrap 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 a 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. CONCLUSION

We have motivated the need for providing a versatile kernel for AOP, as a means to foster consolidation in both use and exploration of AOP. We have identified the main requirements that an AOP kernel should address. We are working on the design and implementation of such a kernel for Java, based on an appropriate extended model of partial reflection and our previous work on Reflex [55].

Acknowledgements

The authors would like to thank Rémi Douence and the reviewers for their highly valuable feedback.
This work is partially funded by Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

5. REFERENCES

[1] M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2004. To appear.

[2] 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*, volume 1603 of *Lecture Notes in Computer Science*, pages 35–49. Springer-Verlag, 1999.

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

[4] 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.

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

[6] L. Bergmans, M. Akşit, 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 AOSD 2004 [3].

[8] N. Bouraqadi-Saâdani, T. Ledoux, and F. Rivard. Safe metaclass programming. In *Proceedings of the 13th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 98)*, Vancouver, British Columbia, Canada, Oct. 1998. ACM Press. ACM SIGPLAN Notices, 33(10).

[9] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the OOPSLA/ECOOP 90 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 303–311. ACM Press, Oct. 1990. ACM SIGPLAN Notices, 25(10).

[10] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In Batory et al. [4], pages 110–127.

[11] D. Caromel, F. Huet, and J. Vayssière. A simple security-aware MOP for Java. In Yonezawa and Matsuoka [60], pages 118–125.

[12] D. Caromel and J. Vayssière. Reflections on MOPs, components, and Java security. In Knudsen [40], pages 256–274.

[13] 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.

[14] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In AOSD 2004 [3].

[15] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2003)*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376. Springer-Verlag, 2003.

[16] The Concern Manipulation Environment website, 2002. http://www.research.ibm.com/cme.

[17] P. Costanza, G. Kniesel, and M. Austermann. Independent extensibility for aspect-oriented systems. In *Workshop on Advanced Separation of Concerns at ECOOP 2001*, 2001.

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

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

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

[21] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. [4], pages 173–188.

[22] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In AOSD 2004 [3].

[23] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In Akşit et al. [1]. To appear.

[24] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [60], pages 170–186.

[25] R. Douence and L. Teboul. A pointcut language for control-flow. In *Proceedings of the 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'04)*, Lecture Notes in Computer Science. Springer-Verlag, Oct. 2004. To appear.

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

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

[28] 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.

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

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

[31] W. H. Harrison and H. L. 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).

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

[33] 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.

[34] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In AOSD 2004 [3].

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

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

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

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

[39] G. Kniesel, P. Costanza, and M. Austermann. JMangler - a powerful back-end for aspect-oriented programming. In Akşit et al. [1]. To appear.

[40] J. L. 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.

[41] 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*, 2003.

[42] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[43] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[44] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A

case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, Feb. 1997.

[45] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed AOP. In AOSD 2004 [3].

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

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

[48] R. Rao. Implementational reflection in Silica. In P. America, editor, *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP 91)*, volume 512 of *Lecture Notes in Computer Science*, pages 251–266, Geneva, Switzerland, July 1991. Springer-Verlag.

[49] A. Rashid. A hybrid approach to separation of concerns: The story of SADES. In Yonezawa and Matsuoka [60], pages 231–249.

[50] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behavior. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in Lecture Notes in Computer Science, pages 205–230, Málaga, Spain, June 2002. Springer-Verlag.

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

[52] D. Sereni and O. de Moor. Static analysis of aspects. In M. Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 30–39, Boston, MA, USA, Mar. 2003. ACM Press.

[53] 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.

[54] É. Tanter, N. Bouraqadi, and J. Noyé. Reflex – towards an open reflective extension of Java. In Yonezawa and Matsuoka [60], pages 25–43.

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

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

[57] M. Wand. Understanding aspects (extended abstract). In *Proceedings of the International Conference on Functional Programming (ICFP 2003)*. ACM Press, 2003.

[58] I. Welch and R. J. 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, July 1999. Springer-Verlag.

[59] 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.

[60] 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.