Metalevel Facilities for Multi-Language AOP

Éric Tanter
Center for Web Research, DCC, University of Chile
Avenida Blanco Encalada 2120, Santiago, Chile
etanter@dcc.uchile.cl

ABSTRACT

Providing metalevel facilities in object-oriented languages has been studied and has resulted in the formulation of a set of design principles advocating the use of mirror-based reflective APIs. In this paper, we explore the provision of metalevel facilities in the context of aspect-oriented programming, more precisely by considering *multi-language AOP*: different aspect languages are available to programmers, in addition to the base object-oriented language. After general design considerations, we discuss the concrete case of Reflex, a versatile kernel for multi-language AOP in Java.

1. INTRODUCTION

1.1 Design Principles: Mirrors

In [2], Bracha and Ungar identify three design principles for reflection and metaprogramming facilities in object-oriented languages:

encapsulation – metalevel facilities must encapsulate their implementation behind interfaces[1];

stratification – metalevel facilities must be separate from base-level functionality;

ontological correspondence – the ontology of metalevel facilities should correspond to the ontology of the language they manipulate.

The authors show that mainstream reflective APIs (*e.g.* that of Java, Smalltalk, C#) do not respect these principles, while APIs based on *mirrors* do. Languages providing mirror-based reflective APIs are Self [13] and Strongtalk [1]. The idea of mirrors is that metalevel facilities are exposed by special objects called mirrors, rather than being accessed from an object or class directly, as in the case in classical reflective APIs:

```
Object o = ...;
Class c = o.getClass();
Class s = A.class.getSuperclass();
```

To ensure encapsulation and stratification, Bracha and Ungar advocate the use of *mirror factories*. A mirror factory is a central "function" to which mirrors are requested:

```
ObjectMirror objM = MirrorFactory.reflect(obj);
ClassMirror clsM = ojbM.getClass();
ClassMirror sclsM = clsM.getSuperclass();
MethodMirror methM = sclsM.getMethod("foo");
```

The problem of classical reflective APIs is that clients are dependent on the implementation details of the reflective system they use. Conversely, with mirror factories, a specific factory can be set according to the situation: it is therefore possible to rely on different implementations of the reflective API transparently. Furthermore, the stratification resulting from the separation of metalevel facilities from base code makes it easier to simply remove reflection support when not used. Doing so can considerably reduce the footprint of an application, for instance. Conversely, if a class that has uses other than reflection holds certain reflective capabilities, then it is hard to safely remove reflective capabilities from the system [2].

Finally, ontological correspondence addresses the symbiosis between the reflective API and the language being reflected upon. It has two dimensions. *Structural correspondence* states that the structure of metalevel facilities should correspond to the structure of the language they manipulate. Such facilities ought to represent both code and computation, and go as deep as within method bodies. Ideally, reflective support should be conditional to what the VM supports and on-demand. *Temporal correspondence* states that the metalevel APIs should be layered so as to distinguish static and dynamic properties of the underlying language. In other words, the distinction that a language makes between *code* (compile time) and *computation* (run time) should be manifest in the APIs.

The objective of this work is to study how the aforementioned principles can be applied in the context of aspect-oriented languages, not only considering a unique language with objects and aspects such as AspectJ [7], but rather considering a context of multi-language AOP, whereby different (possibly domain-specific) aspect languages are used in conjunction with a base object-oriented language.

1.2 Multi-Language AOP

In previous work [10, 11], we have motivated the interest of being able to define and use different aspect languages, including domain-specific ones, to modularize the different concerns of a software system. We have proposed the architecture of a so-called *versatile kernel* for multi-language AOP, and our current Java implementation, Reflex.

An AOP kernel supports the core semantics of various AO languages through proper structural and behavioral models.

[1]Actually, the point is to rely on abstractions rather than concrete implementations – therefore extensible high-level classes are a valid alternative to interfaces.

Designers of aspect languages can experiment comfortably and rapidly with an AOP kernel as a back-end, as it provides a higher abstraction level for transformation than low-level transformation toolkits. The abstraction level provided by our kernel is a flexible model of partial behavioral reflection [12], extended with structural abilities. Furthermore, a crucial role of an AOP kernel is that of a mediator between different coexisting AO approaches: detecting interactions between aspects, possibly written in different languages, and providing expressive means for their resolution.

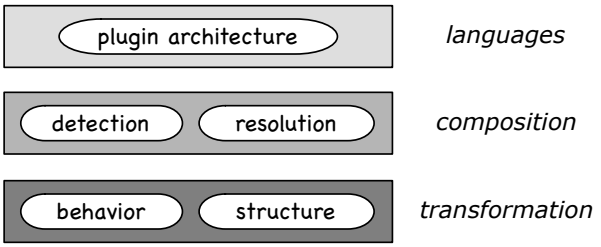

Figure 1: Architecture of a versatile kernel for multi-language AOP.

The architecture of an AOP kernel consists of three layers (Fig. 1): a transformation layer in charge of basic weaving, supporting both structural and behavioral modifications of the base program; a composition layer, for detection and resolution of aspect interactions; a language layer, for modular definition of aspect languages (as plugins).

It has to be noted that the transformation layer is not necessarily implemented by a (byte)code transformation system: it could very well be integrated directly in the language interpreter (VM). As a matter of fact, the role of a versatile AOP kernel is to *complement* traditional interpreters of object-oriented languages. Therefore, the fact that our implementation in Java, Reflex, is based on code transformation should be seen as an implementation detail, not a defining characteristic of the kernel approach.

2. MIRRORS FOR MULTI-LANGUAGE AOP

2.1 Languages Involved

There are different language layers that have to be taken into account when considering multi-language AOP and the corresponding metalevel facilities, as illustrated in Fig. 2:

- **Aspect Languages (ALs)** – These languages are defined by aspect language providers, in order for programmers to be able to express their aspects at the appropriate level of abstraction. Aspect languages can be general purpose or domain specific.
- **Kernel Language (KL)** – The kernel language makes it possible for providers of aspect languages to express aspects in terms of common constructs. The design of the kernel language actually determines the actual degree of versatility of the kernel (*i.e.* the variability of aspect approaches that is supported);
- **High-Level Language (HLL)** – The high-level language is the base (object-oriented) language that is used to develop applications (*e.g.* Java).
- **Virtual Machine Language (VML)** – This language is that natively understood by the execution environment of the HLL (*e.g.* Java bytecode).

Bracha and Ungar motivate the need for explicitly separating VML and HLL when designing metalevel facilities due to the possible discrepancies between both languages [2]. Such discrepancies typically appear when implementing high-level constructs that are not directly supported by the VM. Notable examples in Java are nested classes and generics, which are implemented with synthetic entities that can unfortunately be observed via reflection. Conversely, in Strongtalk [1], the mirror API is divided in two parts: one reflecting Smalltalk, and one reflecting the underlying structures in the VM. This paper does not address the issue of providing metalevel facilities for VML, nor does it go in details of HLL metalevel facilities; we rather focus on KL and ALs, which are specific to multi-language AOP.

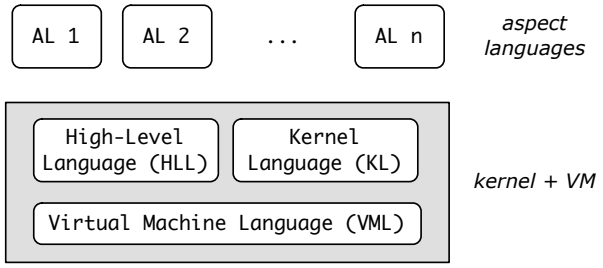

Figure 2: The different languages in multi-language AOP.

2.2 Design Guidelines

We discuss design guidelines for reflective APIs in multi-language AOP by analyzing the consequences of the principles formulated in [2]. When it comes to examples, we stick to an object-oriented HLL since its language concepts are well-known (classes, methods, fields, etc.). We discuss the application of the guidelines distilled in this section to concrete KL and ALs in Section 3, when considering our particular context (Java, Reflex).

2.2.1 Structural Correspondence

The principle of structural correspondence implies that there should be distinct reflective APIs for each language in a system. Therefore, in multi-language AOP, this means that in addition to specific APIs for VML and HLL, the kernel must offer a reflective API for KL, and that *each aspect language must be accompanied by its corresponding reflective API.*

Since the objective of a versatile AOP kernel is to support a virtually infinite set of aspect languages, it is impossible to design the corresponding reflective APIs in advance. At most, a set of guidelines can be provided. This means that the provider of an aspect language plugin must also provide the associated reflective API. Designing languages in tandem with reflection is indeed a healthy exercise: by definition, a reflective API reifies the ontology of a language – if a reflective API is too large and too complex, this can be seen as an indication that the language itself is too large and too complex [2].

2.2.2 Encapsulation and Stratification

Adopting a mirror-based design in order to respect the principles of encapsulation and stratification implies that a particular reflective API is provided as follows:

- a mirror factory gives access to mirror objects;
- mirror objects are handed out by their interface types, not their actual implementation types;
- each concept of the language is reified as a mirror interface.

Furthermore, we identify three responsibilities mirror interfaces should endorse:

- expose the *properties* of the language concept they reify, with possibly getters and setters (*e.g.* the name or visibility of a class or method);
- make it possible to *navigate* in the language concepts organization (*e.g.* extract methods of a class, extract expressions of a method);
- potentially expose a number of *actions* that can be performed on a language concept (*e.g.* instantiate a class, invoke a method, insert code before an expression).

2.2.3 Temporal correspondence

Up to now, we have not taken into account the temporal correspondence principle: mirroring code and mirroring computation should be separable modules of the mirror API [2]. This is necessary because some concepts reified as mirrors or actions available as methods on mirrors require the existence of a running computation: *e.g.* a mirror to an instance of a class, invoking a method or instantiating a class. Hence these mirrors and methods would not make sense in an "offline" use of the API (such as a graphical introspector, like a class browser).

Technically, this separation can be achieved by providing two distinct mirror factories: a *code mirroring factory* and a *computation mirroring factory*. Each factory returns mirrors implementing its own collections of interfaces. For instance, a code mirroring factory would return a class mirror whose interface does not include an `invoke` method; whereas the class mirror returned by a computation mirroring factory would support such a message.

In many cases, most (if not all) of the information available when reflecting upon code is also available when reflecting upon computation. For instance, the name of a class or the set of its methods are available from both a code view and a computation view. One could therefore be tempted to avoid repeating the code API in the computation API. However, doing so has a number of inconveniences: first, users of runtime metalevel facilities would need to handle both APIs, and this would typically require bridges from one world to the other to be provided; second, it is frequent that although the concepts of the code view are repeated in the computation view, the possible usages of properties and actions differ. For instance, in Java, in a compile-time usage of the reflective API, one expects to be able to have strong intercession capabilities, such as changing the name of a class, adding an interface and a bunch of methods to it, etc. Conversely, in a runtime usage, these intercession capabilities may not be provided: in Java, if not running in debugging mode, one cannot change the definition of a class; if running in debugging mode, method bodies can be changed, but the actual structure of a class cannot be changed (*e.g.* by adding new fields).

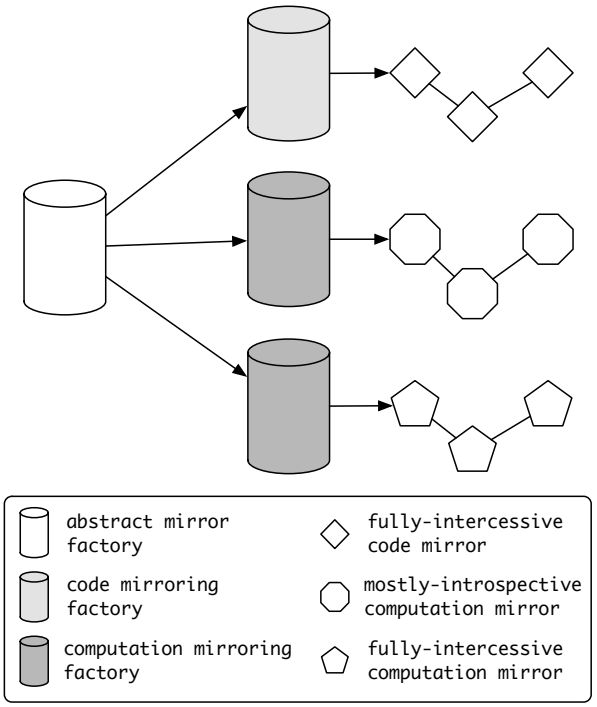

Figure 3: An abstract mirror factory giving access to one code mirroring factory (handing out fully-intercessive mirrors), and two computation mirroring factories: one for debugging, handing out fully-intercessive mirrors, and one for standard execution, handing out mostly-introspective mirrors.

2.2.4 Mirror Factories

Therefore, in order to respect the temporal correspondence principle and support the variability associated to code and computation, a metalevel API should be provided by an *abstract mirror factory* for the language, which gives access to both the code mirroring factory and the computation mirroring factory. Furthermore, *each factory should be self-contained*, in the sense that a user need only interact with a single factory. Note that there might be more than two factories: two computation mirroring factories could be provided, one corresponding to a debugging mode and one for the standard running mode. As a rule of thumb, there should be *as many mirror factories as particular usage scenarios of metalevel facilities.*

3. ILLUSTRATION IN REFLEX

We now discuss the application of the guidelines presented above in the concrete context of Reflex, our versatile kernel for multi-language AOP in Java [11]. We first give an overview of Reflex, and then discuss the design of metalevel facilities for Java, Reflex, and aspect languages (via two concrete examples). We end this section by identifying the need for cross-language metalevel facilities.

3.1 Reflex in a Nutshell

Reflex is a portable library that extends Java with structural and behavioral reflective facilities. Behavioral reflection follows a model of partial behavioral reflection presented in [12]: the central notion is that of explicit *links* binding a set of program points (a *hookset*) to a *metaobject*. Reflex does not impose a specific metaobject protocol (MOP), but rather makes it easy to specify tailored MOPs, which can coexist in a given application.

The aforementioned links are called *behavioral links* to distinguish them from *structural links*, which are used to perform structural reflection. The model for structural reflection is based on the class-object model of Javassist [4, 5]. A structural link binds a set of classes to a metaobject, which can both introspect and modify class definitions (including method bodies).

Reflex is based on Javassist, and therefore operates on bytecode at load time. The transformation process consists, for each class being loaded, of determining and applying first the set of structural links that apply to it, and then the set of behavioral links. During installation of behavioral links, *hooks* are inserted in class definitions at the appropriate places in order to provoke reification at runtime, following the metaobject protocol specified by each link.

3.2 Metalevel Facilities for Java (HLL)

3.2.1 Load-time Reflection

Java as such does not provide any metalevel facilities for manipulating code outside of a running process. The Java reflection API is dedicated to runtime. Javassist [5, 6] is a well-known load-time MOP that complements Java in this regard: one can reflect upon code, with full intercession abilities; furthermore, although the actual transformations are done on Java bytecode, the abstractions provided by the API are at the source code level.

Previous versions of Reflex directly exposed Javassist entities to users. But as a matter of fact, the Javassist API is not compliant with the design guidelines that mirror-based systems promote: reifications are exposed as their implementation classes (*e.g.* `CtClass`, `CtMethod`), *not* via interfaces. This violation of the encapsulation principle became problematic to us at some point, because it made it impossible to ensure structural correspondence appropriately, as will be discussed in Section 3.3. As a consequence, we developed another hierarchy of interfaces that completely hides the implementation dependence on Javassist. Although this new load-time MOP mainly offers the same services as Javassist, it does respect the encapsulation principle: users manipulate interface types, not implementation types.

3.2.2 Runtime Reflection

Java offers an API for runtime reflection, mainly restricted to introspection. Beyond the fact that the possibilities offered by this API are restricted, what is of concern to us in this work is that the reflection API of Java does not follow the design guidelines stated in [2]. First of all, the reflection API is not provided via interfaces but via concrete implementation classes, thereby violating the principle of encapsulation. Second, access to reflective entities is provided by base entities themselves (*e.g.* `o.getClass()`), hence compromising stratification. Furthermore, structural correspondence is not ensured: using the reflection API, one can see synthetic members (generated by the compiler), which are not part of the actual source program, but rather reflect the particular implementation strategy used by the compiler for features that are not supported by the VM (generics, inner classes, etc.).

In the context of Reflex, although we have not (yet) implemented a complete mirror-based reflection API for Java, we were led to do so at least for methods: Reflex includes an interface `Method` (`reflex.api.mop`), whose usage is recommended for implementors of metaobject protocols, instead of the one from `java.lang.reflect`. We exploit this interface in order to provide a consistent view of a reified method in metaobjects in the face of implementation tricks. More precisely, to support reification of the message receive operation, the well-known technique of *method wrappers* is used: the method whose execution should be reified is renamed, and a new method with the original name is added, whose body is the hook to the metalevel [5]. The consequence of this implementation strategy is that a *single* method is implemented with *two* methods in Java: one with the original signature but a synthetic body, and the other with a synthetic signature but the original body.

Introducing an interface for reflecting upon methods makes it possible to hide such implementation strategies: synthetic elements are hidden to the programmer. The result is that the principle of structural correspondence is preserved. To generalize this approach, providing a full-fledged mirror based reflection API for Java is necessary.

3.3 Metalevel Facilities for Reflex (KL)

A domain-specific language for using Reflex with a concrete syntax is currently being designed: until then, the only means to configure Reflex is by using the metalevel facilities of this not-yet-born language: the kernel API. The kernel API makes it possible to define new links, either structural or behavioral, or to introspect existing links. In order to respect the principle of temporal correspondence, a difference is explicitly made in Reflex between the definition of a link, which relies upon and lives at load time, and the runtime representation of a link, which is used *e.g.* to access the metaobject associated to a link.

3.3.1 Load-time Reflection

At link definition time, a link is represented by an object of type `BLink` if it is a behavioral link, or `SLink` if it is a structural link. Both `BLink` and `SLink` are *interface types*, not implementation types: this respects the encapsulation principle. Link objects are never created directly by the programmer, rather they are obtained from a factory:

```
// obtain new link object from factory
BLink link = API.links().createBLink(...);

// set attributes
link.setScope(Scope.OBJECT); ...

// effectively define the link
API.links().addBLink(link);
```

The resulting flexibility of the design is being used in an on-going experiment to extend Reflex to distributed systems: specific implementations of the link interfaces transparently refer to remote links.

Link objects offer full introspective and intercessive abilities. However, due to our implementation approach (bytecode transformation), once a behavioral link is effectively used for the first time, some of its intercessive abilities are disabled. For instance, it is not possible to change the scope of a link if some running objects are already affected by that link: this restriction is made necessary because the scope attribute of a link directly determines how a metaobject reference is implemented (instance variable, class variable, or global). Note that a VM-based implementation of Reflex (or an implementation in a language that fully allows runtime bytecode transformation) could support full intercession on links, even dynamically.

3.3.2 Runtime Reflection

At runtime, a reification of a link is provided via an object implementing the `RTLink` interface. Only behavioral links are reified, since structural links are applied at load time and have no runtime existence (again, this restriction comes from our implementation approach and will to remain portable in the context of Java).

An object or class may be affected by several links. Following the design guidelines of mirror-based systems, one can obtain a reification of the set of links that apply to a given instance by a call to a factory:

```
Object o = ... ;
RTLink[] links = API.links().getLinksFor(o);
```

The reification of a behavioral link makes it possible to dynamically access the metaobject associated to a link, change its activation condition, or introspect its definition (via the corresponding `BLink` object, obtained with `getDefinition()`):

```
Object o = ... ;
RTLink link = ... ;

// getting the metaobject for o
Metaobject mo = link.getMetaobject(o);

// setting a new metaobject for o
link.setMetaobject(o, new A());

// deactivating the link for every objects
link.setActive(Active.OFF);
```

Like classes, links have unique identifiers that can be used to discriminate them. However, our experience with developing Reflex-based tools tends to show that manipulating link identifiers is not necessary in most cases: a proper design will rather make a link reference accessible directly, or hide it behind application-specific abstractions. For instance, if a logging library is developed with Reflex, a global service can be provided to activate/deactivate logging or to change the log level:

```
Object o = ... ;

// deactivating the link "behind the scene"
Logger.deactivateFor(o);

// changing the metaobject "behind the scene"
Logger.setLevel(o, Logger.VERBOSE);
```

This principle of *hiding link manipulation behind abstractions* that are more adequate for the programmer is at the essence of the guidelines for metalevel facilities for aspect languages developed on top of the kernel, discussed in Section 3.4.

3.3.3 Structural Correspondence

A major implication of the principle of structural correspondence is that the metalevel should not make synthetic entities visible. This issue was mentioned previously in the case of the Java reflection API: synthetic members generated by the compiler are unfortunately visible via reflection, whereas they have no relation to the source code; they rather correspond to some implementation strategy of the compiler, to allow HLL to provide some constructs not supported by VML (*e.g.* generics).

In the context of Reflex, the importance of structural correspondence is exacerbated by the fact that all features are implemented via bytecode transformation. In other words, since the execution environment (the VM) is unchanged, the features of KL are implemented as transformations over HLL: references to metaobjects are implemented by synthetic fields initialized by synthetic methods, hooks to the metalevel are implemented by inserting synthetic pieces of code in method definitions, etc.

All these synthetic elements ought to be invisible at all stages. First of all, since the definition of a link (*e.g.* class selectors) relies on code introspection, synthetic members and expressions can mislead the programmer's intention. Second, at runtime, the reflection API of the HLL (Java) should not expose these synthetic elements. For the latter, we already mentioned that the solution lies in a full-fledged mirror-based runtime reflection API. For the former, our solution is to exploit the benefits of our mirror-based version of the Javassist load-time MOP (Sect. 3.2.1).

Since our load-time reflection API is based on interfaces, we are able to provide implementations that are "Reflex-aware" in the sense that they hide synthetic (read, Reflex-generated) members and expressions. In other words, we can ensure that *code modifications done during both application of structural links and installation of behavioral links are not seen by other links*. Furthermore, Reflex provides a means to make this systematic hiding customizable: through a visibility protocol, it is possible for a structural link to expose part of its modifications to either some or all of the other links (details can be found in [9]).

3.4 Metalevel Facilities for Plugins (ALs)

In Reflex, an aspect language, possibly specific to a particular domain, is implemented by a *plugin* [11]. Reflex makes it possible to manually register plugins, and also supports automatic detection of available plugins. In our previous work, we have mainly considered plugins as compile/load-time facilities: a programmer expresses an aspect in a particular aspect language, then the plugin implementing this language is in charge of translating the aspect to the KL. We hereby aim at providing well-structured APIs for runtime reflection too.

The translation of an aspect from a given AL to the KL implies that several *synthetic* kernel entities (*e.g.* links) are generated by the plugin: for instance, a single AspectJ aspect is typically implemented by several links in Reflex [8, 11]. To respect the principle of structural correspondence, it

is important that these synthetic entities be *hidden* to clients of the KL reflective API.

A plugin should provide mirror factories corresponding to the language it supports. Separate factories are required in order to respect temporal correspondence (if it makes sense for the AL in question), and more generally to support the different usage scenarios of metalevel facilities (*e.g.* for debugging), as discussed in Section 2.2.4.

We now briefly discuss the case of two Reflex plugins, the AspectJ plugin [8], supporting a subset of the general-purpose aspect language AspectJ (dynamic crosscutting and composition) [7], and the SOM plugin, supporting a lightweight domain-specific aspect language for *sequential object monitors* (SOM), an abstraction for concurrent programming [3].

3.4.1 Case of SOM

A sequential object monitor (SOM) is an object that is transparently made thread-safe and for which users can define custom scheduling strategies. Under the hood, method invocations on a SOM are reified as requests put in a request queue until scheduled by a user-defined scheduler. A particularity of SOM is its efficient thread-less scheduling mechanism: a SOM is a passive object collaboratively scheduled by client threads.

The SOM DSAL makes it possible to declaratively associate custom schedulers to instances of specified classes:

```
schedule: Dictionary with: ReaderPrioritySched
```

The SOM DSAL is so simple that as of now we have not felt the need to provide a compile-time reflection API for it. However, at runtime, it can be interesting to obtain the scheduler of a particular object, and even to change it. The SOM plugin hence supports a simple runtime API, that follows a mirror-based design:

```
Dictionary d = new Dictionary();

// obtain a SOM mirror on d
SOMMirror m = SOM.reflect(d);

// introspect scheduler of d
Scheduler s = m.getScheduler();
... play with s...

// change the scheduler of d
m.setScheduler(new WriterPrioritySched());
```

Apart from exposing domain-specific concepts to users at the metalevel, the SOM reflection API also ensures that a change of the scheduler is done is a safe manner: the mirror will only change the scheduler when the monitor is free of pending requests. Conversely, giving a direct (read, implementation-level) access to the binding object-scheduler could easily lead to inconsistencies. This illustrates the fact that offering domain-specific metalevel facilities has an advantage beyond the abstraction level offered to programmers: *it makes it possible to enforce that some domain-specific properties are respected while performing metalevel operations.*

3.4.2 Case of AspectJ

The AspectJ plugin (AJP) supports a subset of the AspectJ language, comprising mainly aspects with pointcuts and advices, including `cflow`, advice kinds, `proceed` and reflective join point information.

The standard implementation of AspectJ[2] supports reflection in a way that respects the design guidelines of mirrors: all language-specific concepts (aspects, pointcuts, advices, etc.) are reified as interfaces, instances are obtained via factories, and the temporal correspondence principle is respected by providing two different reflective APIs, `org.aspectj.lang.reflect` and `org.aspectj.runtime.reflect`.

The current reflective APIs of AspectJ are however limited. First, only *introspective* facilities are provided: it could be interesting to extend them with *intercessive* facilities, and to study applications of such new possibilities. Furthermore, the reflective APIs do not support *fine-grained* reification: the insides of pointcut expressions are not reified, nor are advice bodies. Reification of advice bodies would actually present a challenge in terms of ontological correspondence since, in the current implementation, special variables such as `thisJoinPoint` or the special `proceed` statement are translated at compilation time in plain Java. The reflective APIs of our AspectJ plugin for Reflex need to be redesigned in the light of the analysis done in this paper.

3.5 Cross-language Metalevel Facilities

In multi-language AOP, the need for traceability between AL-level entities and KL-level entities arises: as explained in [10, 11], traceability is useful for dealing with detection and resolution of aspect interactions at the appropriate level of abstraction. For instance, although interactions between aspects written in different aspect languages are automatically detected by the kernel at the link level, report to the programmer should be made at the level of the corresponding aspect languages. Furthermore, since KL-level entities generated by ALs are synthetic, they are hidden from clients of the KL reflective APIs. However, it can be interesting to have access to them, *e.g.* for debugging purposes or for building IDE support.

This implies that, on the one hand, given a synthetic KL-level entity, it should be possible to determine the plugin that defined it, and the AL-level entity it represents; on the other hand, given an AL-level entity, the set of synthetic KL-level entities that implement it should be accessible. For kernel-to-AL navigation, we introduced in Reflex the notion of a *linkset*, which makes it possible to embed all synthetic entities corresponding to a single aspect in one kernel entity [11]. Every link contained in a linkset has a reference to it, and this connection is used by the kernel when reporting aspect interactions. Concerning AL-to-kernel navigation, we have implemented a first approach in AJP, whereby a dedicated cross-language mirror factory is used to extract the kernel-level entities corresponding to an AspectJ aspect:

```
// obtain mirror for Foo aspect (AspectJ level)
AJAspect foo = AJ.aspect("Foo");

// get cross-language mirror for foo (AJ-to-Reflex)
AJ2RAspect fooImpl = AJ2Reflex.get(foo);

// get all links implementing foo (Reflex level)
Link[] links = fooImpl.getLinks();
```

[2] `http://eclipse.org/aspectj`

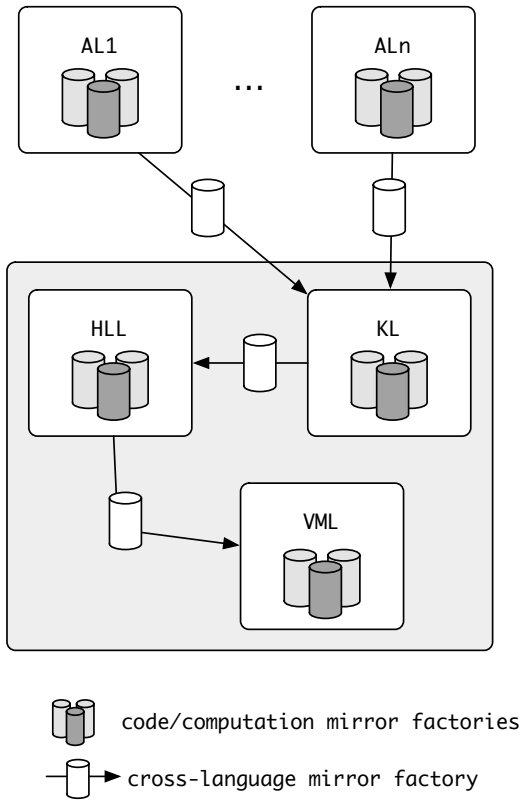

Figure 4: Mirror factories for the different languages, and cross-language factories.

```
// get cflow b-links for pointcut bar
AJPoincut pc = foo.getPointcut("bar");     // AJ
AJ2RPoincut pcImpl = AJ2Reflex.get(pc);    // AJ2R
BLink[] cflows = pcImpl.getCflowLinks();   // R
```

Actually, the interest of providing cross-language metalevel facilities is not restricted to the frontier between ALs and KL, but would also be profitable for the KL-HLL and HLL-VML frontiers (Fig. 4). This is even more necessary if more than one HLL are defined on top of VML. We are not aware of any language proposing these abilities in the case of HLL-VML. Strongtalk [1] is the only language that offers two distinct mirror APIs for HLL and VML, but to the best of our knowledge, it does not provide any cross-language API to trace synthetic entities back and forth.

4. DISCUSSION AND PERSPECTIVES

We have explored how the design guidelines formulated by Bracha and Ungar in their work on mirrors can be applied in the context of multi-language AOP. We have identified that each aspect language should provide its own reflective API in the form of various mirror factories, according to the different usage scenarios of reflection. We have then gone through the case of the Reflex kernel for multi-language AOP in Java. This study has revealed that the Reflex kernel is globally compatible with the advocated design guidelines, but that *(a)* a full-fledged mirror-based API for runtime reflection in Java is required in order to be completely consistent, and *(b)* the metalevel facilities of our AspectJ plugin for Reflex are too ad-hoc and need to be re-designed in the light of this study. Finally, we have identified the interest of cross-language mirror factories, which should be studied further. Another possible alternative for future work lies in easying the task of providing mirror factories for aspect languages. Indeed, a major interest of multi-language AOP consists of being able to define one's own aspect languages; since mirrors reify the concepts of a language, it can be interesting to see how a model-driven approach to aspect language definition can help in generating both parsers and appropriate mirror interfaces and factories.

5. REFERENCES

[1] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 215–230, Washington, D.C., USA, Oct. 1993. ACM Press. ACM SIGPLAN Notices, 28(10).

[2] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 331–344, Vancouver, British Columbia, Canada, Oct. 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[3] D. Caromel, L. Mateu, and É. 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.

[4] S. Chiba. Macro processing in object-oriented languages. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98)*, pages 113–126, Australia, November 1998. IEEE Computer Society Press.

[5] 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.

[6] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In F. Pfenning and Y. Smaragdakis, editors, *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, Erfurt, Germany, Sept. 2003. Springer-Verlag.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[8] L. Rodríguez, É. Tanter, and J. Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV*

International Conference of the Chilean Computer Science Society (SCCC 2004), Arica, Chile, Nov. 2004. IEEE Computer Society Press.

[9] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, University of Nantes and University of Chile, Nov. 2004.

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

[11] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, Lecture Notes in Computer Science, Tallin, Estonia, Sept. 2005. Springer-Verlag. To appear.

[12] 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).

[13] D. Ungar and R. B. Smith. Self: The power of simplicity. In N. Meyrowitz, editor, *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, pages 227–241, Orlando, Florida, USA, Oct. 1987. ACM Press. ACM SIGPLAN Notices, 22(12).