Partial Behavioral Reflection: Spatial and Temporal Selection of Reification

Éric Tanter[1,2] Jacques Noyé[3,2] Denis Caromel[4] Pierre Cointe[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

[4] OASIS project, Université de Nice – CNRS – INRIA – IUF
2004, Rt. des Lucioles, Sophia Antipolis, France

{Eric.Tanter,Jacques.Noye,Pierre.Cointe}@emn.fr — Denis.Caromel@inria.fr

ABSTRACT

Behavioral reflection is a powerful approach for adapting the behavior of running applications. In this paper we present and motivate *partial behavioral reflection*, an approach to more efficient and flexible behavioral reflection. We expose the *spatial* and *temporal* dimensions of such reflection, and propose a model of partial behavioral reflection based on the notion of *hooksets*. In the context of Java, we describe a reflective architecture offering appropriate interfaces for *static* and *dynamic* configuration of partial behavioral reflection at various levels, as well as Reflex, an open reflective extension for Java implementing this architecture. Reflex is the first extension that fully supports partial behavioral reflection in a portable manner, and that seamlessly integrates load-time and runtime behavioral reflection. The paper shows preliminary benchmarks and examples supporting the approach. The examples, dealing with the observer pattern and asynchronous communication via transparent futures, also show the interest of partial behavioral reflection as a tool for open dynamic Aspect-Oriented Programming.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Languages, Experimentation

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.

Keywords

Aspect-oriented programming, open systems, reflection

1. INTRODUCTION

Reflection in programming languages is a paradigm that has emerged from the studies of Brian Smith around the foundations of consciousness and self-reference, as well as his work on the application of these concepts to computer science, down to the implementation of a reflective Lisp dialect [55]. These ideas were then applied to various programming paradigms, including object-oriented programming [40] and had a major impact on languages such as CLOS [32] and Smalltalk [53].

The basic property of reflection is that it supports *metacomputations*, that is, computations about computations. This is done by separating metacomputations and *base* computations into two different levels: the *metalevel* and the *base level*. These levels are *causally connected* [39]. This means that, on the one hand, a reflective program running at the base level has access to its representation at the metalevel, and that, on the other hand, a modification of this representation will affect further base computations. Depending on which part of the representation is accessed, the part describing the (static) structure of the program, or the part describing its (dynamic) behavior, reflection is said to be *structural* or *behavioral*. Another distinction is made between *introspection*, when the representation is simply read, and *intercession*, when the representation is modified.

In this paper, we shall focus on behavioral reflection. The main strength of behavioral reflection is to provide the means to achieve a clean separation of concerns [16, 48], including dynamic concerns, and hence to offer a modular support for adaptation in software systems [3, 52]. These strengths have already been exercised in a wide range of domains, including distribution [6, 36, 42], mobile objects [3, 37, 59], concurrency [41], and fault-tolerance [22].

The applicability of behavioral reflection is, however, limited by the lack of a widely accepted appropriate infrastructure on which to implement reflective applications.

Smalltalk has been considered an interesting basis for building such an infrastructure. It includes quite a number of reflective facilities, it is flexible, and it is implemented in such a way that adding new reflective facilities can be done with very little overhead [20, 23, 53]. However, bringing behavioral reflection into a language such as Java raises new challenges. Indeed, Java only provides very limited reflective facilities and, in order to reduce runtime errors and to take into account security requirements, is much more static in nature. As a result, the overhead due to the additional layer necessary to get behavioral reflection increases significantly. Selecting where and when to apply reflection becomes mandatory. This is called *partial reflection* [29]. Partial reflection makes it possible to balance the effects of compilation, which *embeds* a set of assumptions (a specialization) and reflection, which *retracts* some of these assumptions (a generalization).

As soon as reflection is not native, using such an approach is natural and, therefore, many reflective language extensions do include some kind of partial reflection, where the user can select which entities are reflective (for Smalltalk, see [23, 20, 54], for C++, see [26, 9], for Java, see [11, 60, 51, 56, 52]).

This paper reports on our experiment to provide partial behavioral reflection in a more systematic way. This experiment was performed in the context of Reflex, an *open* reflective extension of Java [56].

The idea of such an open extension was born from the observation that each new reflective extension of Java would provide a particular, monolithic, infrastructure reflecting the commitment of the designer to particular trade-offs, often linked to target application domains, between efficiency, portability, expressiveness and flexibility. In order not to freeze these trade-offs (although we did not sacrifice portability, a major benefit of Java, for efficiency), our suggestion was to introduce a new role in the whole process of developing a reflective application, the *metalevel architect*. This architect, using the framework provided by Reflex, as well as a number of existing building blocks, is responsible for specifying the structure of the metalevel, depending on the requirements of the target application domain. In a standard, closed, extension of Java, the role of the architect is simply played by the designer of the extension. Then, the process proceeds as usual: the *metaprogrammer* implements the building blocks of the metalevel as *metaobject* classes obeying the structure defined by the architect, the *base programmer* develops the base application, and the *assembler* links both levels by implementing the causal connection.

This work includes the following contributions:

- We present a comprehensive approach to partial behavioral reflection, highlighting its spatial and temporal dimensions, and we propose the model of *hook-sets* for behavioral reflection. This model consists of grouping execution points into composable sets, possibly crosscutting the object decomposition, and attaching some metabehavior to these sets through a highly configurable link. The related ideas and techniques are applicable to a wide range of object-oriented languages.

- We describe, in the context of Java, an open architecture supporting this approach and making it possible to combine static and dynamic configuration of reflection. In the context of Reflex, this configuration work is shared between the metalevel architect and the assembler and controlled by the metalevel architect.

- The architecture has been retrofitted into Reflex. Spatial and temporal selection of reflection can be done by combining static and dynamic configuration. Preliminary benchmarks show the benefit of the approach.

- Finally, we illustrate, through examples, how these ideas facilitate the definition of crosscutting metaobjects, clarifying some links between (partial) reflection and Aspect-Oriented Programming (AOP), a paradigm for modularizing crosscutting concerns [34, 21].

The rest of this paper is structured as follows: in Section 2, we introduce partial behavioral reflection in a general setting. Section 3 presents Reflex and preliminary benchmarks. Sections 4 and 5 illustrate how Reflex can be used to achieve separation of concerns on concrete problems, Section 6 discusses related work, and Section 7 concludes with future work.

2. PARTIAL BEHAVIORAL REFLECTION

2.1 Basic principles and terminology

In [43], J. McAffer distinguishes between two approaches to reflection, which consist of either starting from the base-level language elements (e.g., classes), or from the basic *operations* (message send and receive, field access, object creation, etc.) defining the computational behavior of an object. He refers to these two approaches as a top-down and a bottom-up approach, respectively. Actually, one could alternatively refer to them as a structural and a behavioral approach. Indeed, J. McAffer justifies the use of the second approach in a context where he is interested in describing a wide range of object behavior models. As we are interested in behavioral reflection, this is the starting point of our proposal. The metalevel is structured in terms of *metaobjects* reasoning and acting upon *reifications* of the base-level computation described in terms of operations.

In the following, the term *operation occurrence* refers to a particular instance of an operation, further qualified as *static* or *dynamic*, depending on whether the term relates to program text or execution, respectively. *Caller-side* operations occur within method bodies (e.g., message send), and *callee-side* operations at the "surface" of an object (e.g., message receive). Working with Java, new operations may be introduced, such as casts.

A *reflective object* is an object in which some operations are reified and controlled by a metaobject. The link between a base object and a metaobject is called a *metalink*, also referred to as a causal connection link. When an instance of a class is reflective, the class is said to be *reflective*.

2.2 Spatial and temporal selection

Partial behavioral reflection is an approach to more efficient and applicable behavioral reflection that relies on avoiding useless reifications. To this end, high flexibility in specifying reflective needs is required. Partial behavioral reflection addresses the issue of flexibility vs. efficiency by limiting to the greatest extent possible the number of control flow shifts occurring at runtime. Indeed, shifting to

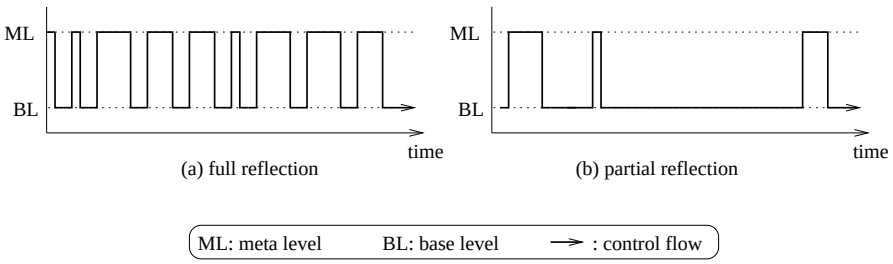

Figure 1: "Reflectogram" of a reflective application: illustration of the evolution of the control flow in a reflective application.

the metalevel is powerful but costly. Such a shift consists of first reifying the operation occurrence, and then delegating (at least part of) its interpretation to the metaobject. In the following, a *hook* is the base-level piece of code responsible for performing a reification and giving control to the associated metaobject. Reification is a significant cause for performance degradation. For instance, in the case of a method invocation, reification usually implies wrapping all the arguments of the invocation into an array of objects and retrieving a reference to a method object. This data may further be encapsulated into a unique method call object. From an efficiency viewpoint, it is therefore crucial to limit the number of shifts and pay the price of reification only when it is effectively needed.

Fig. 1 represents the *"reflectogram"* of a reflective application. A reflectogram illustrates the control flow between the base level and the metalevel during execution. Using full reflection (Fig. 1a), any operation at the base level is reified and therefore many –possibly useless– shifts occur. This does not occur with partial reflection (Fig. 1b). The execution of the application is otherwise unchanged, and hence does not suffer any performance overhead.

We separate two dimensions of the careful selection of reification: the *spatial* dimension, and the *temporal* one.

2.2.1 Spatial selection

Spatial selection consists of selecting *what* will be reified in an application. Spatial selection can be done statically or dynamically. We distinguish between three different levels of selection:

- ***Entity selection*** refers to the selection of the reflective classes and objects. For instance, we may want to specify that classes `A` and `B` are *fully* reflective (all their instances are reflective), and instance `c` of class `C` is reflective (class `C` is *partially* reflective). Other classes and objects are left intact.

- ***Operation selection*** refers to the possibility of selecting which operations are reified for a given reflective entity (i.e., a class or an object). For instance, we may want to reify message receive and field access for class `A`, and only message send for class `B`.

- ***Intra-operation selection*** refers to the possibility of performing fine-grained selection with respect to a particular operation. This selection may be based on characteristics of specific occurrences; for instance, we may want to limit message receive reification to message `foo` in class `A`. For caller-side operations, the selection can also be based on the method/constructor where such an occurrence is found; for instance, we may want to reify only the message sending of `foo` (to instances of class `A`) that occurs in all public methods of `B`.

Intra-operation selection is a crucial property of our approach to partial behavioral reflection. Indeed, it is the finest grain of control over the reification process, making it possible to ignore most of irrelevant operation occurrences during transformation, so that metaobjects do not have to do such selection at runtime.

2.2.2 Temporal selection

Temporal selection consists of selecting *when* reifications are effectively active. It optimizes the overall performance of a system making use of reflection a step further. During the lifetime of an object, the reflective needs may change: a reification may have to be turned off so that metalevel behavior does not apply anymore, or conversely, some external condition may require activating a reification. Obviously, for temporal selection to be worthwhile, the cost of a deactivated reification should be less than that of an activated reification with an empty metaobject. Our benchmarks (Sect. 3.4.5) demonstrate this fact in the context of Java.

2.2.3 A Model of Partial Behavioral Reflection

Spatial and temporal selection consist of precisely selecting the execution points that need to be reified, in order to apply metalevel behavior when needed. The issue is then how to manage such hooks, in order to define how they are linked to the metalevel. Traditionally, behavioral systems adopt a metalink which gathers hooks on a per class or per object basis. However, we feel that such a classical view on metalinks is limited. For the sake of generality, we propose a model of partial behavioral reflection based on the notion of *hooksets*. Hooksets may gather execution points scattered in various objects. This has the nice property of making it possible to apply a metaobject modularly implementing a concern that *crosscuts* the object decomposition. Also, a given object may be involved in several hooksets. This allows for a better modularity of the metalevel, since metaobjects may be assigned a single responsibility. Since the sets of hooks are not restrained to a particular object or class, they need to be named in order to be manipulated.

Hooksets can be composed using the standard set operators: union, difference and intersection. This represents a powerful reuse mechanism for hookset definitions.

In our model, the metalink, to which we will refer simply as *link*, can be described by several attributes:

- the ***scope*** of the link determines whether, for the considered hookset, there is one single metaobject controlling each and every hook (*hookset scope*), or if each class involved has a particular metaobject handling hooks occurring within its instances (*class scope*), or if each object has a dedicated metaobject (*object scope*).
- the ***activation*** of the link is a dynamically-evaluated activation condition that can be specified to achieve expressive temporal selection. The activation condition may be set at various levels (hookset, class, object).
- the ***control*** of the link determines whether the metaobject is given control *before*, *after*, *before and after* an operation occurrence or if it can *replace* it.
- the ***mintypes*** attribute makes it possible to impose type restrictions on the metaobjects that are linked to the hookset.
- the ***updatable*** attribute makes it possible to specify whether the link may be dynamically modified (i.e. the hookset is linked to another metaobject).

In this model, spatial selection is done by defining hooksets, while temporal selection is done through the activation condition of the link. This flexible model is similar to the Event-Condition-Action model used for example in active databases [17]. The event layer is realized by hooksets (hooks are indeed event sources), the condition layer is realized by the activation condition attached to the link, while actions are implemented by metaobjects (Fig. 2).

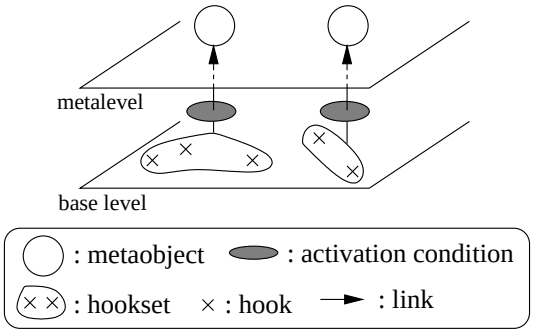

Figure 2: The model of hooksets.

We believe it is important to allow for the *separate* definition of hooksets, metaobjects, and links in order to maximize reuse of both hookset definitions and metaobjects, and to decouple the work of the assembler from that of the metaprogrammer.

2.3 Static and dynamic configuration

As we can see, many aspects of behavioral reflection can be configured. Some of these can be configured statically, and others need to be configured dynamically. For instance, the *assembler* may know that a given reification needs to be activated all the time. On the contrary, a *metaprogrammer* may want to activate some reification on runtime events. As a consequence, both a static configuration API and a runtime API, for dynamic configuration, are needed. The resulting application can then be configured by a combination of static and dynamic configuration.

3. REFLEX: PARTIAL BEHAVIORAL REFLECTION IN JAVA

Reflex is an open behavioral reflective extension of Java supporting partial behavioral reflection, based on the model of hooksets (Sect. 2). The architecture presented here is dedicated to Java, but the concepts and underlying ideas of this architecture are language-independent. Reflex is said to be *open* because, as opposed to other reflective extensions, it does not impose any specific MetaObject Protocol (MOP) [32], thanks to a layered architecture. Indeed, Reflex allows metalevel architects to define their own MOP, based on the framework provided by *Core Reflex*, possibly reusing parts of a *standard MOP* library. This library, built on top of the Core, turns Java into a ready-to-use behavioral reflective system.

In this section, we first introduce the core architecture of Reflex, and then present a standard MOP built on top of it. Based on these elements, we present the interfaces provided by Core Reflex for configuration (both static and dynamic). Finally, we describe some aspects of the implementation of Reflex, and discuss the results of our micro-benchmarks.

3.1 Core Reflex architecture

As motivated at the beginning of the paper, we introduced the role of metalevel architect in the whole process of developing a reflective application. This architect is responsible for defining a specific MOP reflecting a chosen compromise between efficiency, expressiveness and flexibility. This definition is based on Core Reflex.

The overall execution-time picture assumed by Core Reflex is the following (Fig. 3): when a class is about to be loaded, it goes through a selection process whereby Reflex determines whether some hooksets should be associated to the class. If this is the case, the class is transformed into a reflective class before being loaded. Otherwise, the class is loaded as usual.

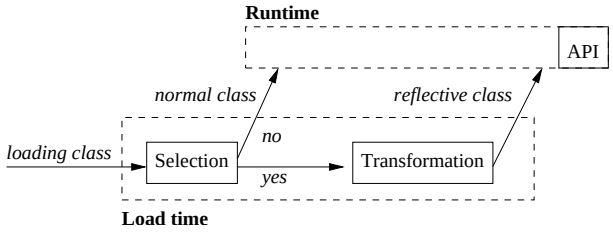

Figure 3: Overview of Core Reflex.

Load-time transformation is discussed in the following section. The selection process is controlled by the assembler: it is driven by defining hooksets, whose definition rely on spatial selection entities (Sect. 3.1.3), and links (Sect. 3.1.5). Temporal selection (Sect. 3.1.4) is initialized statically. A runtime API (Sect. 3.3.2) is provided to the metaprogrammer in order to dynamically control link definition and activation.

The transformation process is controlled by the metaarchitect: the architect defines the available MOP. Defining a MOP basically consists of specifying which operations are reifiable (e.g., message sending), what the interfaces of metaobjects are, and providing code transformation entities able to insert hooks (Sect. 3.1.2). Several MOPs can coexist in a given reflective application.

3.1.1 Load-time bytecode transformation

Because of our concern with the concrete applicability of behavioral reflection, the concretization of our architecture takes the form of a portable Java library. Reflex relies on load-time bytecode transformation, in order to be applicable to binary components and in settings where all classes are not known until they are actually loaded (e.g., open distributed systems)[1].

To make classes reflective, Reflex uses the Javassist framework for load-time structural reflection [10]. Javassist relies on a specific class loader [38]. When requested by the virtual machine, the loader forwards the request to a *class pool*, in charge of locating class definitions. When a class is located and about to be loaded, the class pool can notify a *translator* which can then modify the class. To this end, the translator can obtain reification of classes as `CtClass` objects. `CtClass` offers the same introspection capabilities as those of the standard reflection API of Java, plus intercession capabilities (e.g., adding/modifying a member, changing the superclass, altering method bodies...).

Reflex is connected to Javassist through a particular translator, called a *class builder* (see Sect. 3.4). As a consequence, classes in Reflex are represented at load time as `CtClass` objects.

3.1.2 Defining MOPs

Reflex is an open platform on top of which specific MOPs can be defined. In itself, Reflex does not support any language operation nor does it impose any specific interfaces for metaobjects.

The `OperationSupport` class is used to define the support for an operation in a MOP. An instance of this class encapsulates: a *static operation class*, a convenience name used to refer to this operation, and a *hook installer*.

Operations. The first step when defining a MOP consists of specifying the operations that are supported. For instance, a basic MOP might only offer support for the message receive operation, while a well-furnished MOP might support almost all operations available in the language.

Operations are represented at load time by *static operation classes*. Core Reflex only provides a marker interface for such classes, `StaticOperation`. An instance of a static operation class represents the static occurrence of an operation in a base-level class definition. Hence, by defining operation classes, the metalevel architect controls the exact contents and interface of these load-time representations.

Similarly, so-called *dynamic operation classes* represent language operations during execution. Instances of such classes are *runtime* objects representing an operation occurrence during execution.

Metaobjects. The second step consists of specifying the *interface* of metaobjects, that is to say, what data is reified and how this data will be passed to metaobjects at runtime upon occurrence of language operations. Within the base program, much information is potentially reifiable. This information is not necessarily all of interest to the metalevel architect. For instance, a general-purpose MOP might provide full reifications (including all reifiable data), while a more specific MOP might only provide a simple string description of the operation occurrence.

There are three alternatives for a MOP to pass reified data to metaobjects during execution. Reified data can be passed as a set of arguments, as a single array, or encapsulated within a dynamic operation instance. The decision is driven by a trade-off between abstraction and efficiency.

All these decisions shape the specific interfaces metaobjects are expected to implement. Core Reflex only provides marker interfaces for distinguishing metaobjects: `Metaobject`, and its subinterfaces, `Before`, `After`, `Replace`. Section 3.2.4 presents the hierarchy of metaobject interfaces of the standard MOP.

Hook installers. Hook installers are bytecode transformation entities that have basically two responsibilities: *parsing* a class definition to find occurrences of a given static operation class, and *generating* the appropriate hooks to install, if any. This includes generating the code that will build the reification, and the code that will do the delegation to the metaobject, through the appropriate interface. Reflex provides the `Installer` interface for such entities. They can be implemented using tools such as Javassist, Jinline [58] or BCEL [13].

3.1.3 Spatial selection

In Reflex, spatial selection is done at load time. Each time a class is loaded in a JVM, Reflex determines which hooksets the class is involved in by applying *class selectors*, and then which hooks should be inserted in the class bytecode by applying *operation selectors*. Hooksets are indeed defined *intentionally*, in the sense that the set of points is deduced from the application of predicates (embodied in the selectors).

A **class selector** is responsible for selecting classes (*entity selection*). It implements the `ClassSelector` interface:

```
public interface ClassSelector {
  public boolean accept(CtClass aClass);
}
```

The method `accept` returns true if the class `aClass` should be selected. Having a reification of the class (as a `CtClass`) allows the class selector to select a class on any introspectable characteristics (e.g., the class hierarchy, the parameter types of its public methods, etc.), as illustrated in Section 5.

An **operation selector** is responsible for selecting operation occurrences (*intra-operation selection*). It implements the `OperationSelector` interface:

```
public interface OperationSelector {
  public boolean accept(StaticOperation anOp,
                        CtClass aClass);
}
```

An operation selector can select operation occurrences based on the characteristics of the static operation instance, and on the class to which the selector is applied. For caller-side operations, the operation selector can also limit the

[1] Load-time transformations can also be performed at compile time in order to avoid load-time overhead, provided that classes that need to be processed are available.

scope of reification by selecting operation occurrences based on the specific method/constructor in which they appear.

The use of selectors can be heavyweight if one needs to code specific selectors for each particular case. To facilitate specification of spatial selection, some general-purpose selectors are provided with Reflex, such as the `NameCS` class selector used in the forthcoming examples, which accepts any class whose name was given as a constructor parameter. Providing advanced selectors, able to interpret more expressive parameters (e.g., regular expressions) would be of great value.

3.1.4 Temporal selection

Since spatial selection is done statically based on the program code, we need to provide control for activation at the object level, in order to support reification on a per-instance basis. Furthermore, to be more expressive, activation conditions should possibly be user-defined. Hence, an activation condition may either be a constant, `ON` or `OFF`, or a user-defined condition. Such a condition is an object implementing an interface, `Active`, which declares an `evaluate()` method that receives as argument the current object:

```
public interface Active {
  public boolean evaluate(Object o);
}
```

In addition to providing fine-grained control on activation, it is very convenient to be able to control activation more globally, that is to say, at the class level, and at the hookset level. In order to provide such functionalities in our API, we have two implementation alternatives. The first one consists of keeping only an activation condition in each object, but then to control activation globally we need to be able to retrieve each class involved in a hookset and each instance of a given class. This obviously has a huge overhead in terms of memory usage. Hence, we adopt a solution based on an *activation hierarchy*, in which the hookset level is given priority over the class level, which in turn has priority over the object level. A new condition, `SUB`, is provided to indicate that a given level delegates the responsibility of determining activation. `SUB` is not meant to be evaluated, and is prohibited at the object level, since it does not make sense.

As a result, a hook belonging to hookset hs in a given object o that is an instance of class C will be active if and only if:

- the activation condition of hs evaluates to `true`, or
- the activation condition of hs is set to `SUB` and the activation condition of C evaluates to `true`, or
- both the activation condition of hs and that of C are set to `SUB` and the activation condition of o evaluates to `true`.

3.1.5 Defining hooksets and links

Hooksets. We distinguish between *primitive* and *composite* hooksets (Fig. 4). Every hookset is identified by a unique hookset identifier. A primitive hookset is an operation-specific set defined by a triple *[operation, class selector, operation selector]*, instance of `PrimitiveHookset` (`PH`). Primitive hooksets that relate to the same operation can be composed using the standard set operators (Sect. 2.2.3).

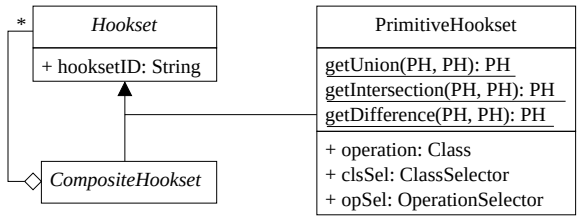

Figure 4: UML class diagram of hookset definitions.

A composite hookset is the union of several hooksets, possibly related to different operations. If an execution point is affected by several top-level hooksets, there is a composition issue. We are working on a convenient way to specify how Reflex should deal with this issue.

Links. Links are described at load time by `Link` objects, and make it possible to specify and characterize the association between hooksets and metaobjects. A link associates a hookset with a *metaobject definition* and a *set of attributes* (Fig. 5).

Metaobject definition. In Reflex, metaobjects are created in a *lazy manner*. This has the advantage of avoiding unnecessary creations (e.g., if a link is never activated), while allowing metaobjects to control the execution of constructors. A `MetaobjectDefinition` object specifies how the metaobject should be obtained. There are basically two means of obtaining metaobjects: either by instantiating a metaobject class, or by using a *metaobject setter* (Fig. 5). In both cases, custom parameters can be specified. They will be passed as an array of strings. A metaobject setter is a kind of factory responsible for initializing the metaobject.

A metaobject setter must define the following static method:

```
void set(Object target, String hsID, String[] args)
```

The `set` method is called whenever the control flow reaches a hook whose metaobject has not yet been set (or has been reset). If the link is of scope object, the `target` argument is the instance to which the metaobject should be linked. If the link is of scope class, then `target` is the concerned class object, and if the link is of scope hookset, `target` is simply `null`. It is of course still possible to create and set the metaobject explicitly without using a metaobject setter.

The `set` method can not only set the metaobject for the object which is passed as parameter, but also, for instance, link this metaobject to other objects, making it a very convenient way to set up an instance-based *crosscutting metaobject*.

Attributes. The various link attributes presented in Section 2.2.3 are available in Reflex. All link attributes are encapsulated within an `Attributes` object (Fig. 5). Control, scope, and updatability are represented by enumeration classes. Mintypes is simply specified as a list of type names. When dealing with composite hooksets, the control attribute may be set to map a particular control value to each composed hookset individually.

Activation support is optional in Reflex. If the activation attribute is `DISABLED`, then the link is not activatable, i.e. it is always active. If activation support is needed, then it is

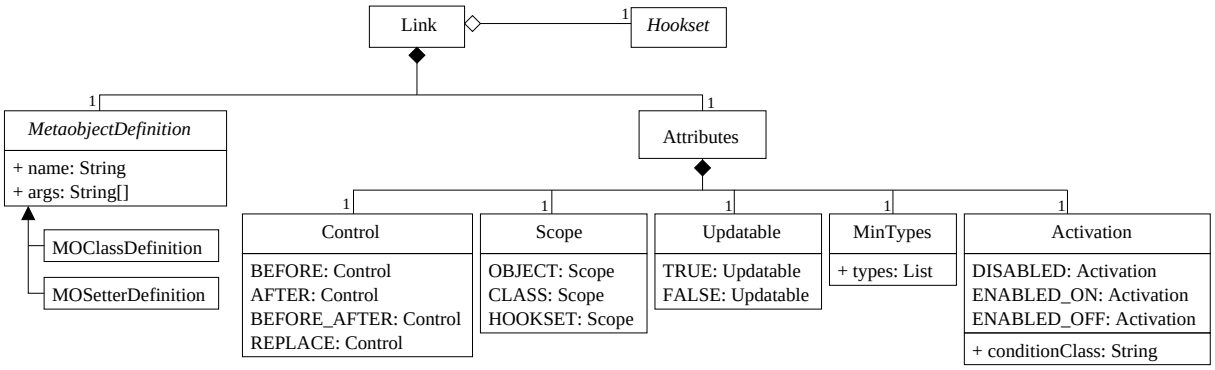

Figure 5: UML class diagram of link definitions.

necessary to specify what activation condition is associated to the link when the application starts:

```
link.setActivation(new Activation("MyCond"));
```

The hookset- and class-level conditions will be set to `SUB` and the object-level condition will be set to an instance of the class `MyCond` (which implements `Active`). For convenience, `ENABLED_ON` and `ENABLED_OFF` are provided. They correspond respectively to `Active.ON` and `Active.OFF` as initial activation conditions.

3.2 Standard MOP

This section describes the standard MOP that is provided by default with Reflex. This MOP is designed to be general-purpose and expressive.

3.2.1 Static operations

Operations are represented by a hierarchy of static operation classes, which are subclasses of the abstract `StdStaticOperation` class (Fig. 6). Each static operation class defines accessors to the various data that can be extracted from the code at load time. For instance, `MsgSend`, which represents the message send operation, defines accessors to the receiver type, the name of the method, the argument types, and so on. Caller-side operation classes have a `getWhere` method to retrieve the method or constructor where the operation occurrence was found, making it possible for operation selectors to refine their selection.

3.2.2 Dynamic operations

Dynamic operations are represented by a hierarchy of dynamic operation classes (Fig. 7). A dynamic operation object encapsulates all the *runtime* information available describing the characteristics of a corresponding operation occurrence.

3.2.3 Static / dynamic operation dependencies

In the standard MOP, when a shift to the metalevel occurs, the target metaobject receives as argument the identifier of the hookset whose member triggered the shift, along with an array of objects referencing the various runtime values of interest. Access to the elements of this array is eased by the fact that a static operation class defines a set of constants (final class variables) that correspond to the indices of the information in the array (Fig. 6).

An instance of a dynamic operation class is an object representation of the array which is passed to the metaobject at runtime. Therefore, a dynamic operation class *depends* on its associated static operation class in the sense that it should define information that maps the information declared in the static operation class. To convert an array into a dynamic operation object, dynamic operation classes implement a static method `convert(Object[])`, which metaobjects can use as follows:

```
class MyMetaobject implements ReplaceCast {
  Object replaceCast(String hsID,
                     Object[] data){
    DCast cast = DCast.convert(data);
    make use of the DCast object
} }
```

This scheme is provided since manipulating dynamic operation instances is convenient but costly. We obtain both flexibility and abstraction (it is easy to get a dynamic operation as an object), and efficiency (one does not pay for it if direct access to the array is enough).

3.2.4 Metaobjects

Metaobjects are instances of classes implementing one of the marker interfaces, `Before`, `After`, and `Replace`, which serve as super interfaces to operation-specific interfaces (Fig. 8).

The method signatures of these interfaces look as follows:

```
void beforeOP(String hsID, Object[] data)
void afterOP(String hsID, Object[] data, Object r)
Object replaceOP(String hsID, Object[] data)
```

Only *replace* metaobjects can change arguments and/or the return value of an operation, while the other types of metaobjects are only able to provoke side effects. Actually, only `replace`*OP* methods have return type `Object`, while the others have return type `void`. In the case of *after* metaobjects, a reference to the return value is passed as an extra parameter.

Since a metaobject class can be multi-operation, it can implement any combination of interfaces. Indeed, dynamic operations are very much like *events*, and metaobjects like *listeners* of various kinds of such events.

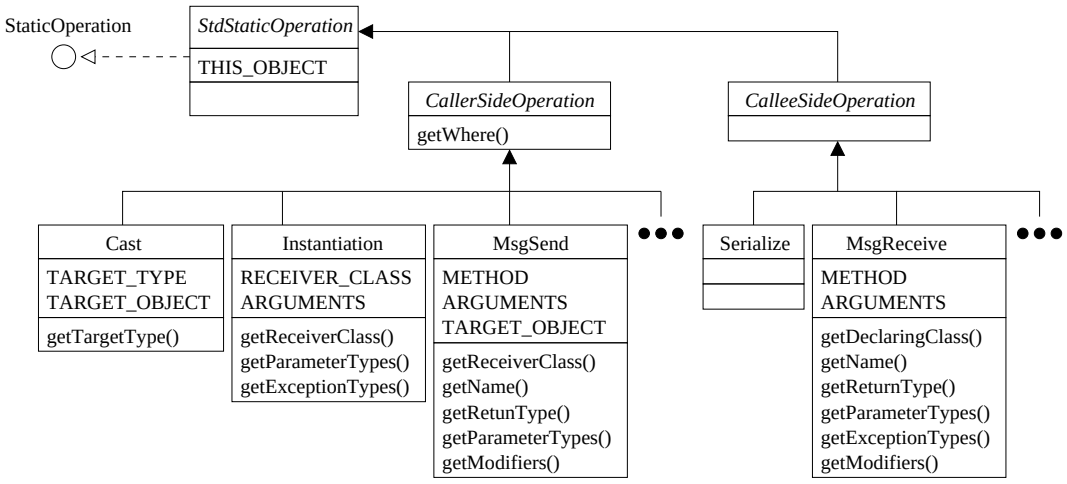

Figure 6: Part of the hierarchy of static operation classes (standard MOP).

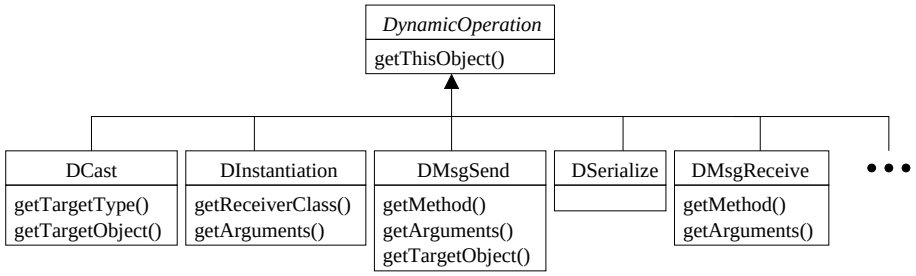

Figure 7: Part of the hierarchy of dynamic operation classes (standard MOP).

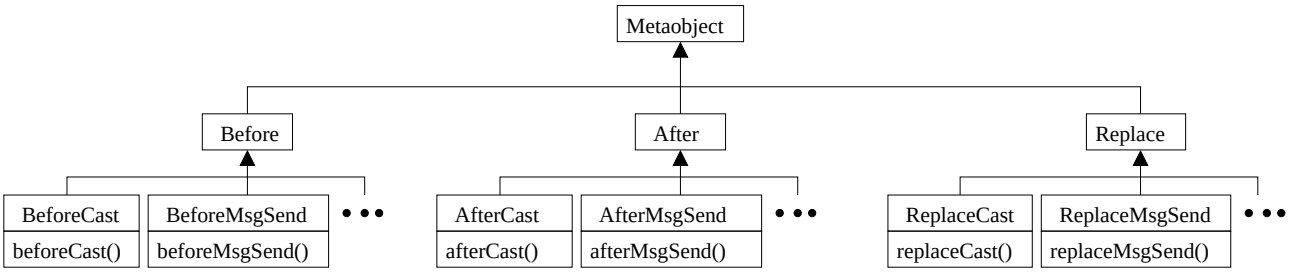

Figure 8: Part of the hierarchy of metaobject interfaces (standard MOP).

3.3 Configuration

Reflex explicitly supports the roles of metalevel architect and assembler. Hence, Reflex provides a dedicated configuration interface for both the meta-architect and the assembler (Fig. 9). Access to these interfaces can be done in various ways, statically and dynamically, as explained later in this section.

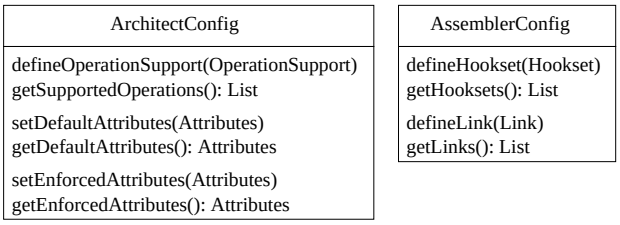

Figure 9: Interfaces for configuration.

In addition to defining supported operations, the metalevel architect can provide default values for the various attributes. He can also restrict the permitted values for the various attributes. For instance, by setting the mintypes attribute, the metalevel architect can enforce the use of a particular metaobject framework. This can be useful to impose a security framework [7], or to provide a common metaobject composition framework [44, 45, 56].

Configuration constraints (e.g., uniqueness of hookset identifiers, respect of enforced attribute values) are checked when new definitions are made; thus these methods throw exceptions in case of violations.

3.3.1 Static configuration

As of now, Reflex supports two forms of static configuration: using *configuration classes*, and using *XML configuration files*. Creating a convenient domain-specific language (DSL) for such configuration is an interesting perspective.

Configuration classes. Configuration of Reflex can be done by providing configuration classes. A configuration class is a class that implements at least one of these two methods:

```
static void initReflex(ArchitectConfig arc)
static void initReflex(AssemblerConfig asc)
```

Reflex keeps a configuration object that configuration classes can fill as needed, using the appropriate interface. All consistency checks are done when defining new elements (operations, hooksets, links, etc.) through the configuration interface. For instance, the following class implements an architect configuration that adds support for the message send operation, using the standard MOP of Reflex:

```
public class SampleArchitectConfig {
  public static void initReflex(ArchitectConfig c){
    OperationSupport msgSendSupport =
      new OperationSupport(
        reflex.std.operation.MsgSend.class, "send",
        new reflex.std.installer.MsgSendInst());

    c.defineOperationSupport(msgSendSupport);
}}
```

Hookset and link definitions can be done in the same manner. The following class illustrates the possibility to embed the hookset and link definitions within a metaobject class (see the discussion on locality in Section 4.2):

```
public class SampleMO implements BeforeMsgSend {
 public static void initReflex(AssemblerConfig c){
   PrimitiveHookset hs =
    new PrimitiveHookset("demo", "send",
                    new NameCS("Foo"),
                    new MsgNameOS("run"));
   c.defineHookset(hs);

   Link l =
     new Link(hs,
              new MOClassDefinition("SampleMO"));
   l.setScope(Scope.HOOKSET);
   l.setControl(Control.BEFORE);

   c.defineLink(l);
 }
 public void beforeMsgSend(String hsID,
                           Object[] data){
   DMsgSend msgSend = DMsgSend.convert(data);
   System.out.println("About to send message: " +
                      msgSend.toString());
}}
```

This class defines a primitive hookset `"demo"` that gathers all calls to a `run` method occurring within a `Foo` class. These operation occurrences will be reified and passed to a unique metaobject (link scope is *hookset*), which will be given control before such occurrences. The defined metabehavior is a simple trace.

XML Configuration files. Reflex also includes an XML parsing module for configuration files. The syntax is a straightforward mapping of the object-oriented configuration presented above. XML configuration is used in the examples of Sections 4 and 5.

Running Reflex. To run an application with Reflex using given configuration files and/or classes, the command:

`java` *Application arg1 arg2...*

should be replaced by:

`java reflex.Reflex -configClasses` *Config1*`:`*Config2...*
`-configFiles` *file1*`:`*file2...*
Application arg1 arg2...

3.3.2 Dynamic configuration

The runtime API of Reflex allows for dynamic configuration by the architect, the assembler, and the metaprogrammer. This API is implemented as a set of static methods of the `Reflex` class.

For the architect and the assembler, the API makes it possible to access the interfaces presented above in this section:

```
ArchitectConfig getArchitectConfig()
AssemblerConfig getAssemblerConfig()
```

All changes made through these interfaces (e.g., newly defined hooksets and links) *only apply to classes loaded afterwards* (see the discussion in Section 3.4.4).

For the metaprogrammer, we can divide the runtime API into three parts, respectively used to control temporal selection, access metaobjects, and create particular reflective instances.

Activation API. This API makes it possible to perform temporal selection. Through the Activation API, it is possible to set and retrieve the activation condition associated to any activatable link, at the desired level (hookset, class, object). The methods to set an activation condition are:

```
setActive(String hsID, Active a)
setActive(String hsID, Class c, Active a)
setActive(String hsID, Object o, Active a)
```

The first parameter represents the identifier of the hookset for which the activation condition should be set. All these `set` methods accept a `null` value as their first parameter: in such a case activation is set for all hooksets. For instance, to deactivate all activatable links, one must call:

```
Reflex.setActive(null, Active.OFF);
```

The `set` methods have `get` counterparts to retrieve an activation condition. A set of logical operators is provided to compose activation conditions as needed.

Metaobject Access API. This API makes it possible to retrieve and set metaobjects at the desired level. To set metaobjects, the following methods are provided:

```
setMetaobject(String hsID, Metaobject m)
setMetaobject(String hsID, Class c, Metaobject m)
setMetaobject(String hsID, Object o, Metaobject m)
```

These methods have `get` counterparts to retrieve metaobjects. The Reflex runtime ensures consistency and enforces the restriction rules that may have been given (e.g., mintypes, updatable).

Creation API. This API makes it possible to obtain a single reflective instance of a possibly already loaded class. This feature is implemented through on-the-fly reflective subclass generation, as presented in [56]:

```
Object createObject(Class c, Link b)
```

This method creates a subclass of `c` and applies the given link to that class. If several links should be given, an overloaded `createObject` method accepts an array of links. Once the subclass has been made reflective, it is instantiated. This approach suffers some limitations: since it is based on subclassing and method overriding, the class cannot be final, and the only operations that can be reified are message receive (for non-final methods) and serialization. Nonetheless, this approach is sufficient for some applications [57].

3.4 Implementation

3.4.1 Class builder

In Reflex, the transformation of a class into a reflective class is controlled by a generic *class builder*. In order to be informed each time a class is about to be loaded, the class builder is connected to a class loader of Javassist [10]. Upon class loading, the class builder queries all class selectors to determine which links have to be applied. The builder then generates some support code within the class, and lets the appropriate hook installer take care of hook insertion for a given operation.

3.4.2 From load time to runtime

Reflex uses a special class loader, provided with Javassist. Recall that class loaders in Java define *namespaces*, which are similar in some sense to small nested logical virtual machines. Reference assignments between namespaces are prohibited [38].

When an application is run with Reflex, the scenario is the following. The `main` method of the `Reflex` class is executed, loaded by the standard class loader (defining $namespace_1$). This method initializes the configuration by invoking configuration classes and/or parsing configuration files. Then a Javassist loader is created so that it uses the Reflex class builder as a translator. This loader defines another namespace, $namespace_2$. The class builder will be notified by the loader of every class being loaded, and accesses the initialized configuration to determine the transformations to apply. In $namespace_2$, a copy of the configuration is created for use by the runtime API.

To sum up, $namespace_1$ is the *transformation space*, where on-the-fly bytecode transformation occurs, and $namespace_2$ is the *execution space*, where the reflective application executes, possibly interacting with the runtime API (Fig. 3).

3.4.3 Support for hook activation

The portable support for hook activation/deactivation is implemented with activation conditions, as presented in Section 3.3.2. Classes and objects involved in an activatable link include an extra field to hold their activation condition. The Reflex runtime stores the hookset-level activation conditions.

An activatable hook differs from a normal one in that a check of the activation conditions (in order: hookset, class, object) is performed before building the reification and invoking the metaobject. Hence, the code of any activatable hook is as follows:

```
if(determine activation) reify and delegate
else original code
```

where the `else` clause is not present for before or after hooks.

3.4.4 Limitations

Our implementation is constrained by the necessity to remain portable and compatible with standard Java. This precludes, at least in a first step, the possibility of resorting to features of the Java Platform Debugging Architecture (JPDA), such as class reloading, which is not meant to be turned on in a production environment. Investigating the benefits of the JPDA in prototyping or debugging environments is certainly valuable and may be part of future work.

Therefore, in the current version of Reflex we are neither able to define new hooksets dynamically for already loaded classes, nor to make activatable a non-activatable hookset. This also explains our implementation of hook activation. In particular, note that, even in the case of a deactivated hook, the activation conditions are still checked repeatedly during execution. *Once loaded in the execution space, class definitions are never altered.*

3.4.5 Benchmarks

We have performed preliminary micro-benchmarks to validate the interest of partial behavioral reflection. The machine used runs under Linux (kernel 2.4), with an Intel Celeron 1.2GHz processor and 760MB of RAM, using the HotSpot client VM from Sun Microsystems version 1.4.1 (build 1.4.1-b21, mixed mode).

The benchmarks are based on the *message send* operation, as implemented in the default MOP provided with Reflex (v1.0 alpha 4). We have measured the execution time of a `loop` method that calls a given number of times a `test` method. Both methods are defined in class `Sample`. In order to associate some computation to the `test` method, the body of this method is a loop incrementing a counter. The aim of the benchmarks is to measure the overhead of applying reflection to calls to the `test` method.

Our micro-benchmark suite consists of the following test cases:

- In the first test case (*non-Reflex*), we measure the execution time of the application without Reflex.
- The second test case (*non-reflective*) consists of running the application with Reflex, including parsing the configuration files and transforming class `Sample` to reify the sending of `test` occurring in `loop`. However, in this case, we use a normal, i.e. non-reflective, object. This lets us measure the sole cost of load-time transformation, without measuring the runtime cost of reflection.
- The following three test cases measure the cost of an empty before/after metaobject respectively when activation is disabled (*bef/aft no act*), when it is enabled and the link is activated (*bef/aft act on*), and finally when activation is enabled but the link is deactivated (*bef/aft act off*).
- We finally considered three similar cases, using replace control instead of before/after (*replace no act*, *replace act on*, *replace act off*). Recall that in these cases, method invocation is done using the Java Reflection API.

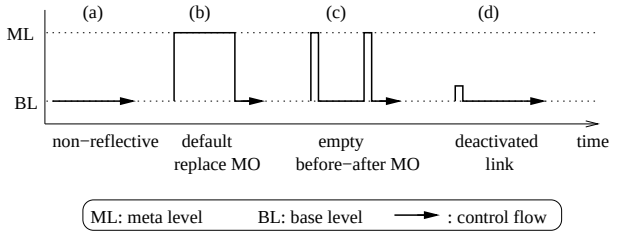

Figure 10: Representation in a reflectogram of the various types of test settings.

Fig. 10 illustrates the various test cases in terms of reflectogram: *non-Reflex* and *non-reflective* are of type *(a)*, *replace no act* and *replace act on* are of type (b), *bef/aft no act* and *bef/aft act on* are of type *(c)*, while the others are of type *(d)*.

We have measured the cost of running the whole application (i.e., starting a new JVM each time) with the Linux `time` command. We have checked that all test cases got a full control of the CPU (99%) and that garbage collection was not triggered.

invocations	1000	5000	10000	20000	50000
non-Reflex	11	37	71	149	401
non-reflective	24	51	85	152	354
bef/aft no act	25	59	102	187	443
act on	26	59	102	188	444
act off	25	55	93	169	396
replace no act	26	61	105	193	455
act on	27	62	106	194	457
act off	25	55	93	169	396

Table 1: Micro-benchmarks results (time in 1/10s).

The two first lines of Table 1 illustrate the overhead of load-time transformation. Plotting these figures gives two parallel straight lines from 1000 to 10000 invocations. The slope of these lines gives the cost of the elementary test: 0.67ms. The value of the ordinates at the origin are 0.4s without Reflex, and 1.8s with Reflex. That is, bytecode transformation has an important impact on start-up time. Partial reflection makes it possible to reduce this impact. For a higher number of invocations, it turns out, quite unexpectedly, that Reflex code becomes more efficient than the Java code. This does not seem to be an epiphenomenon (resulting, from instance, of some HotSpot deoptimization) as the same effect can be observed with interpreted code.

All the other figures correspond to straight lines: there is no runtime disturbance coming from memory management, HotSpot..., and it is therefore easy to compare the various overheads.

- Comparing non-reflective and reflective (replace) execution, we can deduce an overhead of 2 10^{-4}s per reflective invocation. This overhead is indeed pretty high: we have benchmarked standard invocation cost in the same configuration, obtaining a cost of 6 10^{-8}s per invocation. This measurement was done in interpreted mode only, hence it actually represents an upper bound for the invocation cost with HotSpot enabled. The fact that the overhead of reflective invocation is between 3 and 4 orders of magnitude greater than a standard invocation validates the need to precisely select where and when reification occurs.
- The results also show that before/after control is less expensive than replace control with an overhead of 1.8 10^{-4}s per invocation. The gain is however quite moderate. The situation is fairly different in interpreted mode where another series of measurements have shown that the overhead of before/after control was half the overhead of replace control.
- Finally, deactivation is also worthwhile as it reduces the cost of replace and before/after control to 8.4 10^{-5}s (these figures could probably be further improved). On the other hand, one can easily see that the cost of activation is negligible when enabled.

These tests are preliminary micro-benchmarks and therefore call for further measurements. In particular, benchmarking reflection on sizeable applications, while reifying

various operations, would be of great interest. These results may also evolve as we believe that the implementation of Core Reflex can still be improved in a number of ways.

3.4.6 Differences with Reflex 0.3

The version of Reflex presented in this paper is quite different from Reflex 0.3, presented previously [56]. In the old version, there was not a single generic class builder with pluggable hook installers, but instead a hierarchy of class builders, which raised composition issues. Also, the clear separation between Core Reflex and user-defined MOPs did not exist. With regard to configuration, Reflex 0.3 offered limited dynamic configuration API. Extensive static and dynamic APIs are now available. Furthermore, the only means to obtain reflective objects in Reflex 0.3 was through explicit creations (`Reflex.createObject()`), whereas this feature is now enhanced with the possibility of making classes reflective directly through static configuration, hence making it possible to leave the base application code intact. Finally, the model of hooksets as well as the spatial and temporal selection framework were missing.

4. EXAMPLE: THE OBSERVER PATTERN

In this section we illustrate the implementation of the Observer design pattern [24] with Reflex, and contrast it with the corresponding Java and AspectJ [33] implementations, in the line of the work presented at OOPSLA 2002 by Hanneman and Kiczales [28].

The Observer design pattern is one of the patterns that involve crosscutting structures in the relationship between roles in the pattern and classes in each instance of the pattern. As clearly shown in [28], the Java implementation of this pattern leads to mixing functional code of the participating classes with pattern-specific code, which we would like to avoid.

Let us give the Reflex implementation of the Observer pattern for the simple scenario presented in [28]. This scenario includes `Point` objects that are observed by some `Screen` objects. Furthermore, in order to illustrate composability of patterns, some of the screen objects play both the role of *observers* (of the point) and *subjects* for another screen object.

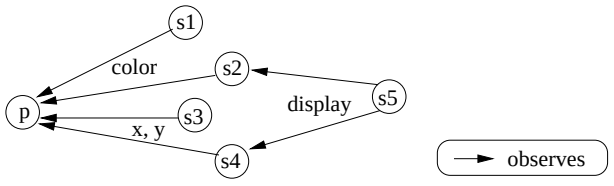

Figure 11: Observation relations between the point object and the various screen objects.

The scenario includes one point `p` and 5 screen objects, `s1`, `s2`, `s3`, `s4` and `s5`. The observation relations are as follows (Fig. 11):

- `s1` and `s2` observe *color changes* in `p`,
- `s3` and `s4` observe *coordinate changes* in `p`,
- `s5` observes *changes in the display* of `s2` and `s4`.

4.1 Implementation with Reflex

4.1.1 Configuration

We have chosen to illustrate the configuration with XML files (Fig. 12).

The architect configuration defines support for the two operations that are used in the present example (*field write* and *message receive*) using the standard MOP, along with default attribute values. Notice that, by default, reifications are activatable with activation initially off.

The definition of three hooksets is required (Fig. 12):

1. `coordinateObserved` applies to instances of class `Point`; notification is triggered upon occurrences of a *field write* operation, where the name of the field is either `x` or `y`.
2. `colorObserved` also applies to instances of class `Point`, and notification is triggered upon occurrences of a *field write* operation, where the name of the field is `color`.
3. `displayObserved` applies to instances of class `Screen`; notification is triggered upon occurrences of a *message receive* operation, where the name of the message is `display`.

4.1.2 Metaobjects

In each of the three hooksets, the metaobject is an instance-specific metaobject that informs the relevant observers *after* the corresponding operation occurrences. To this end, each metaobject holds a list of the observers attached to a particular subject.

The generic part of the pattern behavior (maintaining a list of observers, notifying each of them when needed) has been factored out in the abstract `Observer` metaobject class. This class also offers a service to create observing relations (`map`). The case-specific part of the update logic is left to concrete subclasses: `Observer` defines an abstract `updateObserver` method (Fig. 13).

We define three specific metaobject classes for this example: `CoordinateObserver` and `ColorObserver` (Fig. 13) for observing changes in point coordinates and color, respectively, and `DisplayObserver` for observing display changes on screens.

4.1.3 Running the scenario

The `Main` class runs the proposed scenario by first creating the point and screen objects, and then makes the mapping observers-subject as explained above (Fig. 14). Fig. 15 illustrates the configuration at runtime. For instance, when `p` changes color, this change is reified and passed to metaobject `2` that then notifies each observer (`s1` and `s2`).

4.2 Assessment

The Reflex implementation of the Observer pattern presents the same modularity properties than the AspectJ implementation [28]:

- *Locality* – All of the code that implements the Observer pattern is located in the abstract and concrete metaobject classes; participant classes are left intact. This issue is further discussed below.
- *Reusability* – The core pattern code is abstracted and can hence be reused in other situations. For each pat-

```
<reflexConfig-architect>
  <operations>
    <operation class="reflex.std.operation.FieldWrite" name="fieldWrite"
               installer="reflex.std.installer.FieldWriteInst" />
    <operation class="reflex.std.operation.MsgReceive" name="msgReceive"
               installer="reflex.std.installer.MsgReceiveInst" />
  </operations>
  <defaultAttributes scope="object" activation="enabled-off" control="after" />
</reflexConfig-architect>
```

```
<reflexConfig-assembler>
 <link>
  <hookset id="coordinateObserved" operation="fieldWrite" classSelector="NameCS"
           argsCS="observer.Point" operationSelector="FieldNameOS" argsOS="x | y"/>
  <metaobjectDefinition class="observer.CoordinateObserver"/>
 </link>
 <link>
  <hookset id="colorObserved" operation="fieldWrite" classSelector="NameCS"
           argsCS="observer.Point" operationSelector="FieldNameOS" argsOS="color"/>
  <metaobjectDefinition class="observer.ColorObserver"/>
 </link>
 <link>
  <hookset id="displayObserved" operation="msgReceive" classSelector="NameCS"
           argsCS="observer.Screen" operationSelector="MessageNameOS" argsOS="display"/>
  <metaobjectDefinition class="observer.ScreenObserver"/>
 </link>
</reflexConfig-assembler>
```

Figure 12: XML configuration of the metalevel architect (*top*) and the assembler (*bottom*) for the Observer design pattern implementation.

```
public abstract class Observer {
 protected List itsObservers;
 public void addObserver(Object o){ itsObservers.add(o); }
 public void removeObserver(Object o){ itsObservers.remove(o); }
 protected void updateObservers(Object aSubject){
   Iterator theIter = itsObservers.iterator();
   while (theIter.hasNext()) updateObserver(theIter.next(), aSubject);
 }
 protected abstract void updateObserver(Object aObserver, Object aSubject);

 public static void map(String hsID, Object s, Object o){
  ((Observer) Reflex.getMetaobject(hsID, s)).addObserver(o);
  Reflex.activate(hooksetID, s, Active.ON);
}}
```

```
public class ColorObserver extends Observer implements AfterFieldWrite {
  public void afterFieldWrite(String aHooksetID, Object[] aReifiedData, Object aRetValue){
    updateObservers(aReifiedData[0]);
  }
  protected void updateObserver(Object aObserver, Object aSubject){
    ((Screen) aObserver).display("Screen updated because color changed.");
}}
```

Figure 13: Metaobject classes for the Observer design pattern example.
The `Observer` class is generic and reusable, while the `ColorObserver` is specific to the example.

```
public class Main {
 public static void main(String[] args){
   Point p; Screen s1, s2, s3, s4, s5;
   create the objects
   Observer.map("coordinateObserved", p, s1); Observer.map("coordinateObserved", p, s2);
   Observer.map("colorObserved", p, s3); Observer.map("colorObserved", p, s4);
   Observer.map("displayObserved", s2, s5); Observer.map("displayObserved", s4, s5);
   play with p
}}
```

Figure 14: Main program for the Observer design pattern example.

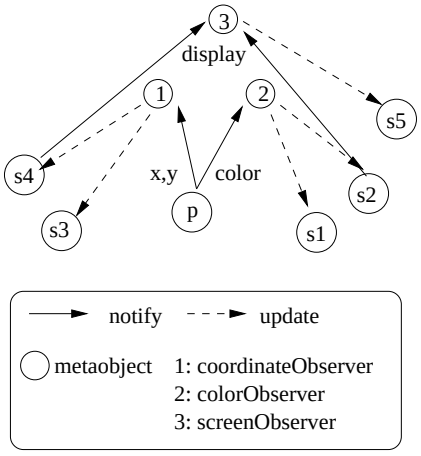

Figure 15: Illustration of the configuration at runtime for the Observer scenario.

tern instance, we only need to define concrete metaobject classes and set configuration as required.

- *Composition transparency* – Since there is no coupling between a participant and the pattern, a subject or observer can take part in multiple observation relations.
- *(Un)pluggability* – Since participants are not *aware* of their role in a pattern instance, it is possible to switch between using and not using a pattern.

With regards to locality, it is important to notice that both the Reflex implementation and the AspectJ implementation rely on an abstract class/aspect, three concrete subclasses/aspects, and a dynamic configuration to setup the scenario. The major difference between Reflex and AspectJ is that, with AspectJ, the pointcut declaration and the link are embedded within the aspect definition. Although this can be seen as a nice property, it is also an example of *tangled* implementation: defining a set of points, a behavior, and the link between them are *different concerns* that should possibly be specified separately. Reflex allows for such separation, which enhances hookset and metaobject reuse. In cases where this level of separation is not needed, metaobject classes can embed their configuration (playing the role of configuration classes, as in Section 3.3.1), which is then equivalent to the AspectJ case.

Furthermore, Reflex preserves a pure object-oriented implementation style: it uses metaobject classes and configuration classes, both of which are implemented as standard Java classes. The configuration files are optional declarative alternatives to configuration classes. Conversely, aspects in AspectJ *are not* standard classes. Hence, Reflex can directly benefit from advanced support for Java development (e.g., incremental compilation in the Eclipse IDE[2]).

From a functional point of view, (un)pluggability can be taken a step further in Reflex than in AspectJ since the support for link (de)activation makes it possible to switch off/on the use of a pattern dynamically. Moreover, the sources of the underlying events can be (de)activated locally (per hookset/object) or globally very easily.

5. EXAMPLE: TRANSPARENT FUTURES

We now show how to use Reflex to implement an important feature of distributed object systems that offer asynchronous communication: transparent futures (see for instance ProActive [6]). This example illustrates the use of runtime activation/deactivation, the need to control the cast operation, and the interest of our selection framework.

5.1 Transparent futures

Futures were first introduced in MultiLisp [27]. In a distributed object system with futures, a call between two processes returns immediately. The client process does not need to wait for the value returned by the server process (unlike, for instance, Java RMI). The returned object, called a *future*, is in fact just a *place holder* for the actual result. While the server process executes, the client can continue its activity, and may even pass the future by reference to other objects/processes. The client process must block and wait for the server process to terminate only when the future is effectively accessed. This is called *wait-by-necessity*.

5.2 The problem of futures with simple MOPs

Using a runtime MOP is a common and easy way to implement transparent futures, as done in ProActive. A future is in fact a reflective object whose metaobject implements the wait-by-necessity strategy. However, apart from the well-known identity issue (the future *is not* the result), this approach suffers some limitations in a strongly-typed language like Java, due to the widespread use of downcasts. A simple MOP supporting only message receive is not sufficient. Indeed, when the future is created, the declared type

[2] `www.eclipse.org`

of the result is known, but not its runtime type. This implies that the future might not be of an adequate type when downcasted later on.

Let us suppose that class `B` is a subclass of class `A`. Consider the following method `foo` of a class `Server`:

```
public A foo(){
  ...
  finally returns a B object
}
```

When this method is called asynchronously, the system creates a future, type-compatible with `A`, and returns it to the caller. With typical client code such as:

```
A a = Server.foo();
...
((B) a).m();
```

an exception will occur on trying to downcast the future before the result has been received.

5.3 A solution with Reflex

The downcast problem presented above can be solved using an expressive MOP supporting cast in addition to message receive, as does the standard MOP provided with Reflex. Also, the expressive power of the selection framework of Core Reflex as well as the activation scheme provided are of great help in this case. This section explains how to design the solution, and the following one shows how to configure Reflex in order to implement it.

Figure 16 illustrates the situation at the time `foo` is called (Fig. 16(a)), corresponding to the classical scenario of transparent remote calls with a runtime MOP, and once the future has been returned (Fig. 16(b)). At this time, the client and the future share a common metaobject, the future metaobject. This metaobject controls both *casts* occurring in the client and *message receptions* occurring in the future. While the result has not been delivered yet (delivery is done by the proxy metaobject, Fig. 16(b)), if the future metaobject intercepts either a cast of the future within the client or a message reception on the future, then execution is blocked until the result is available.

Once the result has arrived, two cases have to be considered:

- the result is of type `A`: the metaobject fills the future with all the fields of the result so that the future becomes (a clone of) the result. At this time, controlling message reception on the future is no longer needed, hence the future is deactivated (using `Reflex.deactivate(future)`). From now on, the future is a standard object.
- the result is of type `B`: the metaobject keeps a reference to the result. Then, each time it intercepts a message receive on the future, it forwards it reflectively to the result. Later, if it intercepts a cast by the client, then it returns the result instead of the future, and execution proceeds.

5.4 Implementation

The first step consists of defining the desired `future` hookset. This hookset is defined as the union of two primitive hooksets (Fig. 17):

- `futureReceives` is a set including all calls to public methods of future classes. This set is defined by a simple operation selector selecting all public methods for the message receive operation (`PublicOS`), and a class selector matching all future classes (`FutureCS`).
- `clientCasts` is a set including all casts to a future type (`FutureCastOS`) occurring in any class (`AnyCS`).

We then define a link between the `future` hookset and the metaobject class `FutureMO`. The link is given hookset scope, so that only one shared instance of `FutureMO` is created. The metaobject class implements both `ReplaceCast` and `ReplaceMsgReceive`. It holds a table of associations between asynchronous call identifiers and future objects.

```
<hookset id="future">
 <hookset id="fReceives"  operation="MsgReceive"
     classSelector="FutureCS"
     operationSelector="PublicOS" />
 <hookset id="clientCasts"  operation="Cast"
     classSelector="AnyCS"
     operationSelector="FutureCastOS" />
</hookset>

<link hookset="future">
 <metaobjectDefinition class="FutureMO"/>
 <attributes control="replace" scope="hookset"/>
</link>
```

Figure 17: XML configuration for the future example.

The definition of `FutureCastOS` does not present any particular difficulty. In contrast, the definition of the `FutureCS` is not easy. Recall that the role of this class selector is to select the future classes. A straightforward solution consists of selecting any class that is a possible result type (or subtype) of a public method of a class on which an asynchronous call is performed[3]. Obviously, in such a case, `FutureCS` might end up selecting almost all classes in the system (for instance, if a method has return type `Object`).

We are actually facing a trade-off between transparency and partiality. Fortunately, Reflex makes it possible to choose and implement the appropriate trade-off depending on requirements. The straightforward solution is totally transparent, but it may lead to a heavy use of reflection. A more elaborate alternative can be based on some heuristics or static analysis to determine the future classes. Finally, one can initialize `FutureCS` with a list of future classes names, or make all future classes implement a marker interface. This last alternative is definitely not transparent, but is optimal in terms of partial reflection.

5.5 Assessment

This example shows a concrete case where controlling the cast operation turns out to be interesting. Furthermore, link activation is very useful here, since we are facing a scenario where beyond a certain point in time reification is not needed any longer.

This example also illustrates the interest of a powerful selection mechanism, as opposed to *type patterns* in AspectJ, and other purely syntactic-based approaches. Let us consider, for instance, the class selector we mentioned as a possi-

[3]This is a class of *active objects* in ProActive.

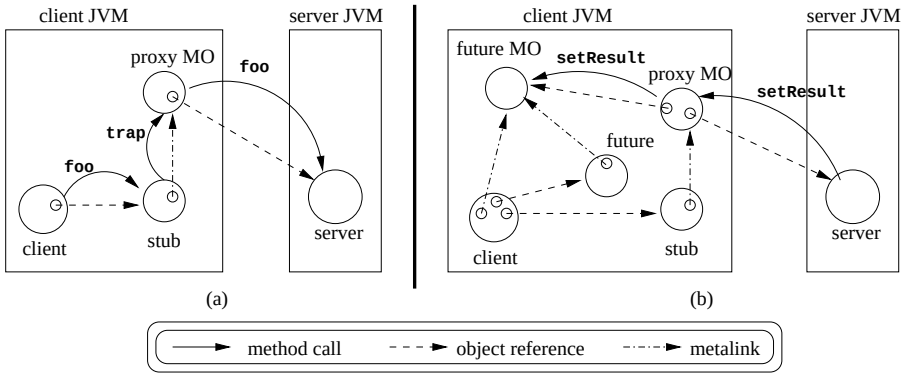

Figure 16: Illustration of the future scenario. (a) The client calls `foo` on the server. (b) The future has been returned to the client and later, the result is delivered to the future metaobject.

ble alternative, which precisely selects those classes that are *the result types of the public methods of the classes on which asynchronous calls are performed*. Such an advanced criterion cannot be expressed using patterns (even compound ones) on type names.

Also, the possibility of adopting various selection strategies (based on syntax, program analysis, introspection, etc.) is a great advantage in terms of flexibility since system designers are not constrained by a particular, closed, way of specifying selection. In the future example, the adopted solution eventually depends on the particular requirements of the target distributed object systems in terms of transparency and partiality.

6. RELATED WORK

AOP has found inspiration in several research efforts, including reflection and metalevel programming, composition filters [2], and multi-dimensional separation of concerns [47]. The question of the relation between reflection and AOP therefore naturally arises. G. Kiczales defined AOP as a *principled subset of reflection* [31]. We first review existing tools and proposals in the fields of reflection and AOP, and then we discuss the relation between reflection and AOP in general, and between partial behavioral reflection and dynamic AOP.

6.1 Existing tools and proposals

6.1.1 Reflective systems

There are several behavioral reflective systems that can be considered to provide some sort of *limited* partial behavioral reflection: Reflective Java [61], Dalang and Kava [60], MetaXa [25], Guaraná [45], and Iguana [26], first in the context of C++, and its Java incarnation, Iguana/J [52]. Apart from Iguana, all these systems only provide the possibility to choose which classes are made reflective.

The Iguana approach is the closest to ours in its philosophy. Compared to other proposals, it adds the possibility of grouping metaobject classes into *protocols* and then selecting the classes on which these protocols do apply. A limitation of Iguana is that it makes it possible to restrict the set of reified operations, but not to extend it. Furthermore, it does not support intra-operation selection, which is a crucial feature of partial behavioral reflection, as presented in this paper. Iguana/J [52] is an implementation of the Iguana model for Java. With respect to temporal selection, it is more powerful than Reflex since it does not require hooks to be specified and introduced at load time. However, this is only possible because Iguana/J is implemented as an extended JVM, hence restricting portability. Note that MetaXa and Guaraná also rely on modified virtual machines. Conversely, the objective of Reflex is to promote the use of behavioral reflection in real-life applications, for which portability is a key issue.

As for configuration, no other reflective architecture offers the same range of static and dynamic configuration possibilities. For instance, Kava only supports static configuration, while Iguana/J only supports dynamic configuration. With regard to supporting combinations of metaobject control, similarly, we are not aware of any reflective architecture that provides such flexibility. For instance, Kava only offers before/after control to metaobjects, while Iguana/J only offers replace control. The explicit support for the role of metalevel architect is also a distinguishing property of our work. Finally, the mentioned systems all adopt a classical view on the metalink and have not considered the handling of crosscutting metaobjects, which is made possible in Reflex thanks to the model of hooksets.

6.1.2 AOP systems

There is a wide spectrum of approaches to AOP, from completely static to completely dynamic, as very well discussed in [52]. In this spectrum, Reflex can be characterized as supporting dynamic binding at load time, meaning that bindings (i.e., hook introduction) are made at load time and can be undone/redone at run time (i.e., changing metaobjects, deactivating links).

JAC [49] is a dynamic AOP system based on a reflective infrastructure set up at load time. PROSE [50] is a dynamic AOP system based on both a modified JIT compiler and a modified JVM. PROSE deliberately sacrifices portability to achieve high performance. These approaches, as well as AspectJ, provide fixed languages for pointcut definitions, mainly syntactic-based, and support a fixed and limited number of base operations.

Event-based AOP (EAOP) [19] is an approach to dynamic aspect-oriented programming, based on the concept of a central *monitor* that receives synchronous *events* from

join points in the application and possibly triggers some actions on sequences of related events. The great expressive power of EAOP makes it possible to reason about *event patterns*, thus supporting temporal reasoning. EAOP can be seen as a particular instantiation of partial behavioral reflection, where all hooks forward control to a unique omnipotent metaobject called the monitor. The most interesting part of EAOP is the way the "metalevel" is structured around advanced event pattern matching facilities. These facilities contribute to raising the level of abstraction, with explicit means of composing aspects, and facilities for the detection and resolution of conflicts between aspects [18].

6.2 Reflection and AOP

One of the strengths of AOP, at least as initially formulated, is that it provides the developer with domain-specific languages, or, let us say, aspect-specific languages (ASLs). Because an ASL is specific to a particular domain or aspect, it fits better the needs of the specialized programmers. The right concepts can be directly manipulated, additional support provided in terms of verification, debugging... Conversely, reflection is a general conceptual framework that can be used to cleanly modularize concerns, but it forces users to understand and assimilate the metalevel viewpoint on the computation. In this sense, reflection is *solution-oriented* since it is expressed in terms of the solution space (the operations of the language used to build the solution), whereas AOP is *problem-oriented* since it is expressed in terms of the problem space (the abstract specification of the problem). However, focusing on ASLs further complicates aspect interaction and composition. Therefore much of the research effort around AOP has shifted to general approaches. The evolution of the AspectJ [33] language is a good example of this trend.

Runtime AOP is a subset of partial behavioral reflection. Building a general-purpose aspect language on top of a generic reflective system makes it easier to provide guarantees in terms of aspect behavior and to lower the complexity of programming (i.e., by "hiding the meta"). The issue is then how to develop specific languages on top of such generic infrastructures. Reflective approaches *can* support AOP development using multiple aspect languages. Aspect-Oriented Logic Meta Programming (AOLMP) [15, 62] is a proof of feasibility. AOLMP offers a general declarative framework for programming aspects (the base level can be a Java or Smalltalk program, while the metalevel is programmed in a logic programming language). This framework can be used to declare rules about interactions and compositions of aspects defined in ASLs [4].

Finally, it should not be forgotten that AOP only aims at handling crosscutting concerns modularly, whereas reflection in general, and behavioral reflection in particular, has several other application areas, such as dynamic programming, generic programming and program adaptation where some kind of self-reasoning is important.

7. CONCLUSION AND FUTURE WORK

We have presented a comprehensive approach to partial behavioral reflection, based on the hookset model, with its spatial and temporal dimensions. Then we have described a behavioral reflective system for Java that fully supports our model in an open and portable manner. Reflex seamlessly integrates load-time and runtime behavioral reflection.

The main contributions of this work are:

- presentation of the model of hooksets, which generalizes the classical view of metalinks, offering support for relations between several execution points and capacity to handle crosscutting concerns,
- full integration of many reflective aspects in a generic model and an open implementation,
- clearly separated specification of hooksets, links and metaobjects, which can still be embedded in one single place,
- intentional, possibly declarative, description of the MOP entry points, using an expressive reification selection framework (class and operation selectors),
- open MOP support, which provides extensibility in terms of supported operations, and flexibility in the way they are reified,
- high level of configurability, both at load time and runtime,
- explicit support for the roles of assembler and metalevel architect in addition to that of the metaprogrammer.

As far as Core Reflex is concerned, interesting perspectives include the possibility of more precisely specifying the needed content of reifications, and the extension of Reflex in order to support MOP generation. Also, we plan to study the interest, in terms of performance, of providing metaobject inlining, and to study how the Reflex implementation could be optimized.

Other future work includes further testing (and possibly extensions) of the capabilities of Reflex through its application to demanding domains such as concurrent programming, distributed systems (some preliminary work has been done in adaptable mobile systems [59]) and context-aware applications [5, 8, 14]. We would also like to analyze the possibilities offered by a metaobject composition framework [44, 45, 56] when combined with a security framework for reflective applications [7].

We have started working on an integration of the Event-based AOP model on top of Reflex. The objective here is to explore the continuum between reflection and aspect-oriented programming. We are particularly interested in the possibility of keeping this infrastructure open but still providing its users with structure and guidance as well as guarantees on the resulting programs (for instance, in terms of safety or security).

Finally, a more distant perspective is to study the instantiation of the model of hooksets in a dynamically-typed object-oriented language like Smalltalk.

8. ACKNOWLEDGEMENTS

We would like to thank Noury Bouraqadi, Shigeru Chiba, Pierre-Charles David, Julia Lawall, and Patricio Salinas for their comments on previous versions of the paper, as well as the anonymous reviewers who have helped us enhance both our work and the paper in a number of ways.

This work was partially funded by Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile, the CONICYT-INRIA ProXiMoS project, and the RNTL ARCAD project.

9. REFERENCES

[1] 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.

[2] L. Bergmans. The composition filters object model. In *Proceedings of the RICOT symposium on* Enabling Objects for Industry, June 1994.

[3] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, and K. Saikoski. A principled approach to supporting adaptation in distributed mobile environments. In *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 3–12, Limerick, Ireland, 2000.

[4] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages. In Batory et al. [1], pages 110–127.

[5] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. In Yonezawa and Matsuoka [63], pages 126–133.

[6] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, Sept. 1998.

[7] D. Caromel and J. Vayssière. Reflections on MOPs, components, and Java security. In Knudsen [35], pages 256–274.

[8] K. Cheverst, C. Efstratiou, N. Davies, and A. Friday. Architectural ideas for the support of adaptive context-aware applications. In *Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality*, Bristol, UK, Sept. 2000.

[9] S. Chiba. A metaobject protocol for C++. In OOPSLA95 [46], pages 285–299.

[10] 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.

[11] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In O. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, volume 952 of *Lecture Notes in Computer Science*, pages 482–501, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[12] P. Cointe, editor. *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection '99)*, volume 1616 of *Lecture Notes in Computer Science*, Saint-Malo, France, 1999. Springer-Verlag.

[13] M. Dahm. Byte code engineering. In C. Cap, editor, *Proceedings of JIT'99, Berlin*, pages 267–277, 1999.

[14] P.-C. David and T. Ledoux. An infrastructure for adaptable middleware. In R. Meersam and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, ODBASE 2002*, volume 2519 of *Lecture Notes in Computer Science*, pages 773–790. Springer-Verlag, Oct. 2002.

[15] K. De Volder and T. D'Hondt. Aspect-oriented logic meta-programming. In Cointe [12], pages 250–272.

[16] E. Dijkstra. The structure of *the* - multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[17] K. R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of ADBMS features. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985, pages 3–20. Springer-Verlag, 1995.

[18] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. [1], pages 173–188.

[19] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [63], pages 170–186.

[20] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming*, June:39–50, 1999.

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

[22] J.-C. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud, and Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 489–498, Pasadena, CA, USA, June 1995. IEEE Computer Society Press.

[23] B. Foote and R. Johnson. Reflective facilities in Smalltalk-80. In N. Meyrowitz, editor, *Proceedings of the 4th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, pages 327–335, New Orleans, Louisiana, USA, Oct. 1989. ACM Press.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.

[25] M. Golm and J. Kleinöder. Jumping to the meta level, behavioral reflection can be fast and flexible. In Cointe [12], pages 22–39.

[26] B. Gowing and V. Cahill. Meta-object protocols for C++: The Iguana approach. In Kiczales [30], pages 137–152.

[27] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[28] J. Hanneman and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, pages 161–173, Seattle, Washington, USA, Nov. 2002. ACM Press.

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

[30] G. Kiczales, editor. *Reflection'96*, San Francisco, CA, USA, Apr. 1996.

[31] 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.

[32] G. Kiczales, J. Des Rivières, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[33] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In Knudsen [35], pages 327–353.

[34] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. 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*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[35] 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.

[36] T. Ledoux. OpenCorba: a reflective open broker. In Cointe [12], pages 197–214.

[37] T. Ledoux and M. N. Bouraqadi-Saâdani. Adaptability in Mobile Agent Systems using Reflection. RM'2000, Workshop on Reflective Middleware, Apr. 2000.

[38] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 36–44, Vancouver, British Columbia, Canada, Oct. 1998. ACM Press.

[39] P. Maes. *Computional reflection*. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium, 1987.

[40] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[41] H. Masuhara, S. Matsuoka, and A. Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, volume 94-PRG-18, 1994.

[42] J. McAffer. Meta-level programming with CodA. In W. Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 190–214, Aarhus, Denmark, June 1995. Springer-Verlag.

[43] J. McAffer. Engineering the meta-level. In Kiczales [30], pages 39–61.

[44] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In OOPSLA95 [46], pages 316–330.

[45] A. Oliva and L. E. Buzato. The design and implementation of Guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems (COOTS'99)*, pages 203–216, San Diego, CA, USA, May 1999.

[46] *Proceedings of the 10th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, Austin, Texas, USA, Oct. 1995. ACM Press.

[47] 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.

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

[49] R. Pawlak, L. Seinturier, L. Duchien, and G. Floring. JAC: A flexible solution for aspect-oriented programming in Java. In Yonezawa and Matsuoka [63], pages 1–24.

[50] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: Efficient dynamic weaving for Java. In M. Aksit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 100–109, Boston, MA, USA, Mar. 2003. ACM Press.

[51] B. Redmond and V. Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. ECOOP 2000 Workshop on Reflection and Metalevel Architectures, June 2000.

[52] 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.

[53] F. Rivard. Smalltalk: a reflective language. In Kiczales [30], pages 21–38.

[54] F. Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Université de Nantes, École des Mines de Nantes, June 1997. In French.

[55] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Jan. 1984.

[56] E. Tanter, N. Bouraqadi, and J. Noyé. Reflex – towards an open reflective extension of Java. In Yonezawa and Matsuoka [63], pages 25–43.

[57] E. Tanter and J. Piquer. Managing references upon object migration: Applying separation of concerns. In *Proceedings of the XXI International Conference of the Chilean Computer Science Society (SCCC 2001)*, pages 264–272, Punta Arenas, Chile, Nov. 2001. IEEE Computer Society.

[58] E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In Batory et al. [1], pages 283–298.

[59] E. Tanter, M. Vernaillen, and J. Piquer. Towards transparent adaptation of migration policies. In *8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002)*, Málaga, Spain, June 2002.

[60] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Cointe [12], pages 2–21.

[61] Z. Wu. Reflective Java and a reflective component-based transaction architecture. In J.-C. Fabre and S. Chiba, editors, *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in Java and C++*, Oct. 1998.

[62] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[63] A. Yonezawa and S. Matsuoka, editors. *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, Sept. 2001. Springer-Verlag.