Reflex — Towards an Open Reflective Extension of Java

Éric Tanter[2], Noury M. N. Bouraqadi-Saâdani[1], and Jacques Noyé[1]

[1] École des Mines de Nantes
La Chantrerie - 4, rue Alfred Kastler
B.P. 20722
F-44307 Nantes Cedex 3
France
{Noury.Bouraqadi, Jacques.Noye}@emn.fr
[2] University of Chile
Faculty of Physics and Mathematics
Computer Science Department
Av. Blanco Encalada 2120, Casilla 2777
Santiago, Chile
etanter@dcc.uchile.cl

Abstract. Since version 1.1 of the Java Development Kit, the Java reflective facilities have been successively extended. However, they still prove to be limited. A number of systems (e.g. MetaXa, Guaraná, Kava, Javassist) have addressed this limitation by providing reflective extensions of Java with richer MetaObject Protocols (MOPs). All these extensions provide a particular infrastructure that reflects the commitment of the designer to particular trade-offs between efficiency, portability, expressiveness and flexibility. Unfortunately, these trade-offs are not satisfactory for all applications, since different applications may have different needs. This calls for breaking down the building of a reflective extension into different components that can be specialized in order to fit specific needs. We qualify such a reflective extension as *open*. In this paper, we present Reflex, a prototype *open reflective extension of Java*. As such, Reflex is a working reflective extension implemented by composing basic building blocks organized following a framework. Reflex comprises the definition of the framework, default generic components and some specialized components.

1 Introduction

Our initial objective was to apply a reflective extension of Java to enhance mobile agent systems with regards to the way the resources attached to a mobile agent are handled upon migration (see [23]). Using a reflective extension in such an application domain implied several strong requirements such as portability and the ability to attach a metaobject to only some specific instances of a given class.

When looking around for appropriate reflective extensions, we could not find one that would fit our needs. We therefore started implementing our own simple

extension based on Javassist [5]. In this extension [23], a reflective object was attached to a unique metaobject which understood a single MOP (MetaObject Protocol) method for trapping method invocations. That is, the metaobject was activated on each invocation of a public method of its reflective object through *hooks* introduced via code transformation. These hooks were looking as follows:

metaobj`.trapMethodcall(`*args*`);`

Later on, we discovered that, in some cases, it was necessary to give control to metaobjects when their base object was being serialized. However this feature was not offered by our simple reflective extension. The code transformation process was extended in order to add to each reflective class a method of the Java serialization API, `writeReplace`, automatically invoked when serialization occurs, which was made to invoke another method on the metaobject, in the following way:

metaobj`.trapSerialize(`*args*`);`

This means that the MOP was extended with a new method. The annoying part of this was that the previously developed metaobjects were not compatible with the new MOP, since they did not implement the new method. It was all the more annoying as this extended MOP was only required for some particular objects and metaobjects.

In fact, the issue we encountered there is a recurrent one. On the one hand, there are *high-level* reflective extensions providing hardwired choices about MOP definition and hook introduction, as well as about some important trade-offs such as performance vs. portability. What happens then if these choices are not compatible with the application requirements? On the other hand, there are *low-level* byte-code manipulation APIs allowing the definition of a custom-built reflective extension, at some non-negligible development cost. There is no middle ground, no reflective extension that would both limit the number of hardwired choices and allow seamless customization and extension in order to suit the requirements of a particular application or class of applications.

This paper suggests that providing such a reflective extension, which we arguably qualify as *open*, is a worthwhile task. It presents Reflex, a prototype *open reflective extension of Java*. Reflex is a working reflective extension implemented by composing basic building blocks organized following a framework. Reflex comprises the definition of the framework, default generic components and some specialized components. This introduces, besides the classical roles of metaobject programmer and end-user, a new role in the development of a reflective application: the *architect of the metalevel*, who is responsible for defining, based on the framework as well as a number of existing building blocks, a fully-defined reflective extension.

The main ideas on which Reflex is based are the definition of a generic MOP and the reification of the code transformation process as an extensible entity.

The idea of a *generic MOP* replaces the idea of a global, all-encompassing, MOP since needs in this regard are unpredictable. It is of course possible to

offer a large MOP, but it will never cover all possible needs. In general, a MOP method is devoted to handle a particular type of *event* occurring at the base level, like method invocation, serialization, creation... Even if these types of event could be completely identified, the way each type of event has to be dealt with cannot be predicted. For instance, in the case of method invocation, there is a possibly infinite set of ways to deal with it: one could like to handle accessor methods in a particular way (therefore requiring a method like `trapAccessor`), or to handle methods distinctly, depending on some *method categories* (hence requiring methods like `trapMethodCategoryA`,...). This is why Reflex is based on a generic and minimal MOP consisting of a single method, called `perform` due to its similarity with the *perform* method of Smalltalk. This method takes as its first argument a string describing the event. Metaobject invocations, and therefore hooks, look as follows:

metaobj`.perform(`*event*`, `*args*`);`

To put the generic MOP into practice, these hooks must be inserted where needed. The corresponding code transformation is reified as an extensible entity, which we call a *class builder*, in order to set up the appropriate hooks within a given class, using *subclassing* when the class cannot be modified.

The following section, Sect. 2, comes back on the requirements that should be met by an open reflective extension. Section 3 describes Reflex, a first step towards such an extension, details its main concepts, the generic MOP and class builders, its architecture, and shows how it meets the above-mentioned requirements. Section 4 provides some examples that illustrate the use of Reflex. Section 5 discusses related work, Sect. 6 future work and Sect. 7 concludes.

2 Requirements

This section reviews the basic requirements in terms of portability, expressiveness, and efficiency which led to the design of Reflex. Apart from the fact that portability is not compromised over, great care is taken not to discard any option too early.

2.1 Portability

A major benefit of Java is its *portability*. In our opinion, this major benefit should not be lost when considering a reflective extension. It would be somehow contradictory to provide an extension which would include portability restrictions! We have previously mentioned that the initial target application of Reflex, mobile agents, requires portability. We expect many applications of a reflective extension of Java to share such a requirement.

This discards the idea of relying on a specific virtual machine or just-in-time compiler, as considered by systems such as Guaraná [18] or MetaXa [13], as well as extending the VM with native, and therefore platform-dependent, code

through the Java Native Interface (JNI), as in Iguana/J [19][1]. This also means that the hooks intercepting base level events should be introduced through code transformation (either byte-code or source-code transformation). Moreover, in order to be 100% Java compliant this transformation should be restricted to application code. Java core classes should be kept untouched.

2.2 Expressiveness of the MOP

Let us first consider how the link between the base and the meta-level is handled. In the following, we shall refer to this link as the *metalink*. For the sake of generality, we shall assume that the metalink is instance-based (rather than type-based), has cardinality n-n (that is, a metaobject can be associated to several objects and, conversely, an object to several metaobjects), and is dynamic, making it possible to dynamically adapt the behavior of a base object.

A second important issue is the definition of the base events. A quick review of the literature on the applications of reflection (see, for instance, [3, 20, 17]) shows that a simple MOP providing only control of method invocation covers the needs of a large range of applications. However, application developers may need metaobjects that control other events than method invocation (e.g. object creation), or may need to handle these events in an adapted manner (e.g. for performance enhancements, or to introduce method categories). We have previously mentioned our need to control object serialization.

This means that the architect of the metalevel should be able to define new kinds of events together with the corresponding hooks. This includes the definition of the hooks, the definition of the code transformation responsible for their introduction and the time when this code transformation takes place. Note that, in a given application, the set of events of interest may vary, at least from one type of object to the other, if not from one object to the other. In the latter case, the introduction of different hooks should nevertheless keep the objects *type-compatible*. Actually, the set of events of interest could even vary along the lifetime of an object. We shall assume here that all the potential interesting events are known when the hooks are introduced. However, we do not assume any specific time (compile time, load time or object creation time) for hook introduction. This means that a standard, *non-reflective*, object (an object which is not hooked to the metalevel) may be turned into a *reflective* one.

Finally, expressiveness also covers metaobject composition, i.e. assembling the metaobjects attached to a given base-level object. Let us note that this goes beyond the provision for a metalink of cardinality n-n as soon as the different metaobjects associated to a given base object interact. A *composition policy* describes the strategy according to which metaobjects are assembled. As there is no universal composition policy, an open reflective extension should make it possible to support different composition policies and allow the definition of new policies. For the sake of reusability, composing metaobjects designed to handle different sets of events should be possible.

[1] The case for rejecting this possibility is actually weaker and would require a more thorough discussion.

2.3 Efficiency

The use of reflection introduces two different kinds of overhead. There is one kind of overhead which is due to the introduction of reflection, more precisely the introduction of hooks in base-level code. There is another kind of overhead which is due to the execution of the hooks. Paying these costs for any class and object would be an overkill. There is therefore a basic rule which says that these costs should only be paid for classes and objects that require it. This has a number of implications:

- Reflective and non-reflective objects may live together within an application. The execution of the standard objects should not be affected by the presence of the reflective objects. Note that, with respect to the previous discussion on portability, this requirement is difficult to guarantee when modifying the standard Java compilation and run-time support.
- It should be possible to apply reflection on an instance basis. Care must be taken to keep the reflective and non-reflective instances of a given base class type-compatible.
- Even if the interesting hooks are known at object creation time, it makes sense to delay their introduction so that a non-reflective object which is bound to become a reflective one is not slowed down until really needed. Conversely, if the hooks are not needed any longer, it would be nice to be able to transform the reflective object back into a non-reflective one. The decision of delaying hook introduction may depend on the respective overheads of executing the hooks (with a useless indirection via a dummy metaobject immediately transferring the control back to the base object) and introducing the hooks after object creation time.
- It should be possible to choose to perform hook introduction at compile time (dealing with source code or byte-code) so that the cost of hook introduction is not incurred at run time (if it matters), or even to merge the base level and the metalevel as is done in the so-called compile-time MOPs [6], when both levels and their links are known at compile time. Much more aggressive optimization techniques could actually be envisionned, based on run-time code generation [10] and partial evaluation [14, 2], but being able to seamlessly combine well-known compile-time and run-time reflection techniques would already be quite a progress.

3 The Reflex framework

Reflex is a prototype working open reflection extension fulfilling the requirements presented in section 2. It defines a framework for its components that can be specialized to meet particular needs. The already implemented components respect this framework and can either be extended or simply replaced with other components fitting into the framework. Such concrete components include hook introduction for the generic MOP (see sections 3.1 and 3.2) and specialized

MOPs (e.g., the one presented in section 4), and metaobjects implementing a composition scheme (see sections 3.4 and 4).

The Reflex package (with all source code) available for download at the Reflex website [24] comes along with several sample metaobjects and programs.

This section focuses on the APIs and abstractions provided to the architect of the metalevel. The upper part of Fig. 1 shows the main elements of the Reflex framework. We shall come back to the lower part of the figure, showing an actual specialization, in the next section. These elements are:

- `ReflexMetaobject`: an interface defining the basic services provided by a metaobject,
- `ReflexClassBuilder`: an interface defining the basic protocol for hook introduction,
- `Reflex`: a class that allows triggering code transformation (statically or dynamically) and attaching metaobjects to base-level objects, and
- `ReflexObject`: an interface that defines the protocol to access the metaobjects linked to a given object. Class builders are responsible for making *reflective classes*, i.e. classes defining reflective objects, implement this interface.

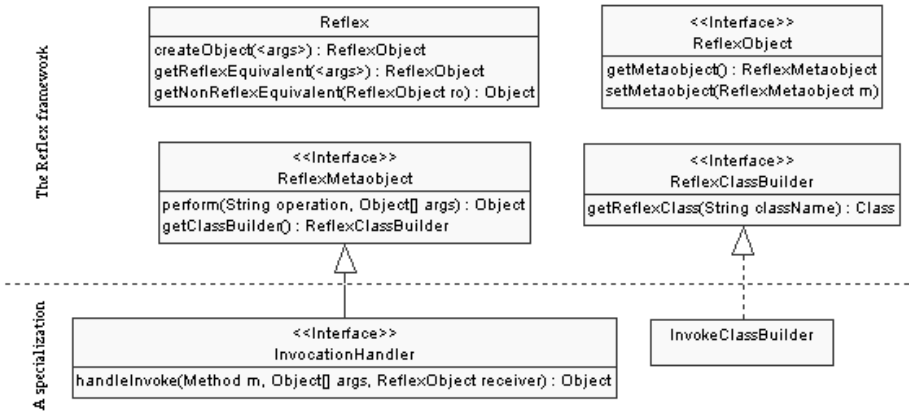

Fig. 1. The Reflex framework and one specialization

3.1 The generic MOP

Fig. 1 shows that, by default, a metaobject implements the interface `ReflexMetaobject`, which defines two methods. The Reflex package includes a default implementation of such metaobject. The method `getClassBuilder` returns the class builder implementing hook introduction (see below) as required

by this metaobject. A hook is an invocation of `perform` with two arguments: the event type and an array of additional parameters depending on the events.

As an example, let us consider hooks for method invocations. In a more "traditional" MOP, trapping a method invocation would be done using the following hook:

metaobj`.handleInvoke(`*method*`,` *args*`,` *receiver*`);`

While, with Reflex, trapping the same method invocation can be done using the following *generic* hook:

metaobj`.perform("handleInvoke",` *invokeArgs*`);`

This hook could result, for instance, in the invocation of a method `handleInvoke` of the metaobject. The argument *invokeArgs* describes the base-level invocation trapped by the hook: method, arguments, and receiver.

Such a MOP makes it apparent that a metaobject *interprets* base-level events. It is very flexible in that new event types can be introduced without redefining the MOP (including generic class builders), which also means that metaobjects defined in such a context are quite easy to reuse even in the presence of new events. We shall also see in section 3.4 that it may also help when implementing a composition policy.

There are however some drawbacks associated to this flexibility. First, the use of a generic hook has some cost (an additional indirection with a dispatch on the event type and some argument packing and unpacking) which may turn out to be prohibitive. Second, the choice of a uniform metaobject type may be counterproductive in a context where many *intended* types of metaobjects are manipulated, with the risk of preventing early type mismatch detection.

This could indeed be a serious problem if Reflex were a closed reflective extension, which it is not. The architect of the metalevel may choose to extend the MOP, as will be illustrated in the next section.

3.2 Class builders

Class builders are responsible for hook introduction. A class of class builders implements the `ReflexClassBuilder` interface (see Fig. 1), i.e. a class builder is able to take as input a standard class and return a *reflective* class. This reflective class can then be instantiated in order to get reflective objects which are type-compatible with instances of the initial class.

Therefore, a class builder corresponds to the set of events attached to the input class. It operationally defines these events as a program transformation introducing the proper hooks at the proper places in the code. Depending on the class builder, this code may be either source code or byte-code and the transformation may happen in place (the input class is destructively turned into a reflective class) or not. In the latter case, *subclassing* should be used in order to get type compatibility between instances of the input class and the output class. Different class builders corresponding to different events can then produce different compatible reflective classes that can coexist in a running system. Class

builders can therefore be built according to the precise requirements (performance, dynamic adaptability, events of interest...) of the target applications. They play an essential role with respect to the flexibility of the framework.

A class builder may first generate an implicit subclass of the original class if needed (using a `SubclassCreator` object). It then performs a sequence of transformational operations on this class. Such transformations are reified as entities, defined in classes such as `MetalinkInserter` to insert the metalink in the class, `MethodcallHookInserter` to wrap methods so that shifting to the metalevel is done, `MethodCopier` to copy compiled methods, etc. These transformation components are implemented with the Javassist API [5] and therefore operate on bytecode.

For instance, the generic class builder available in the Reflex package allows metaobjects to trap, through the generic MOP, invocations of public methods. This builder first creates a subclass, overrides all inherited public methods, and wraps them using the generic hook (`perform`) to give the control to the metalevel. It then inserts the metalink (reference to a metaobject). Then the generated subclass can be compiled on disk or not, before being returned for instantiation. Defining a new class builder is a matter of extending this predefined process (e.g., by subclassing) or defining a new one (for instance to avoid the generation of the subclass).

Any reflective class, created via a class builder, implements the `ReflexObject` interface, which defines the `getMetaobject` and `setMetaobject` methods for respectively retrieving and setting the metaobject attached to a base reflective object. This means that, in this still-imature version of Reflex, the metalink has cardinality n-1, i.e. a base object is linked to a single metaobject (though several metaobjects can cooperate using a composition scheme).

Note that there is another classical way of introducing hooks: using object wrappers. This is the technique used in Dalang [26] and the dynamic proxy classes offered by JDK1.3 [15]. Such a scheme suffers from a number of well-identified problems [26], in particular from the identity problem and the lack of type compatibility between a standard, non-reflective, object and its wrapper, making it reflective. These problems can be circumvented either by merging the object and its wrapper, as in the successor of Dalang, Kava [26], or by making the wrapper inherit from the wrapped object, as in the MOP used by ProActive [4]. In the former case, the issue is then to combine reflective and non-reflective instances of the same class. [26] mentions the introduction of hooks on the sender's side, without more details. In the latter case, there is the cost of an additional object.

The idea of relying on inheritance in order to control invocations is not new [9, 11]. As for Java reflective extensions, the idea has already been used by Reflective Java [27]. However, Reflective Java does not provide any creation protocol (see below), which requires, when programming at the base level, to know the reflective classes attached to a base class and to manipulate these classes explicitly.

3.3 Creating reflective objects

The `Reflex` class (see Fig. 1) defines the *creation protocol*, i.e. static methods dedicated to creating reflective objects as well as turning non-reflective objects into reflective objects (and vice-versa).

The `createObject` methods are used to create a reflective object from scratch. These methods should be used instead of `new` statements by the application programmer when a reflective version is needed[2]. The `createObject` method exists in different overloaded versions. The first argument is always the name of the class to be instantiated (with the idea that, behind the scene, a reflective version of it is going to be used). One version of the `createObject` method takes a second argument which is the metaobject to attach to the created reflective object. As previously seen, the metaobject gives access to the class builder to use to create the appropriate reflective class in case this class has not been constructed yet. Another version takes as second argument the class builder. The metaobject is then a default metaobject which does not modify the behavior of the object.

The `getReflexEquivalent` methods are used to build a reflective instance from an existing, non-reflective, one. As, in Java, there is no way to dynamically change the class of an object, this reflective instance is a shallow copy[3] of the non-reflective instance. The `getReflexEquivalent` method exists as well in different overloaded versions. The only difference with the `createObject` methods is that the first argument is an object (the one to get a reflective equivalent of) instead of a string representing the name of a class. A dual `getNonReflexEquivalent` method makes it possible to produce, again through cloning, a non-reflective object from a reflective one.

3.4 Composition framework

Reflex does not enforce the use of a particular composition scheme, nor does it enforce the use of a composition scheme at all. However, to simplify the creation of such composition schemes, the Reflex framework includes a generic composition framework.

This framework simply makes explicit the distinction that we see between three types of metaobjects, composers, extensions, and interpreters:

- A *composer* acts as a *facade* [12] of the set of composed metaobjects. It defines the composition policy and is in charge of managing the construction and evolution of the composition set.
- An *interpreter* defines a complete meaningful interpretation of base-level events (e.g., method invocation). A remote call metaobject is an example of an interpreter.
- An *extension* simply extends the interpretation of such events with some extra behavior. A trace metaobject is an example of an extension.

[2] Obviously, this is not needed if the original class has been directly modified, may it be at compile time or load time.

[3] The values of the object fields are not copied recursively.

The generic composition framework therefore consists of three empty interfaces, `Composer`, `Interpreter`, and `Extension`, deriving from the root interface `ReflexMetaobject`. Section 4 mentions a concrete composition scheme deriving from this framework.

A composer should be as generic as possible, hence it should make as few assumptions as possible on the types of the metaobjects it will compose. Therefore, since metaobjects in the composition set will not be retrievable directly through the metalink, and may offer public methods for configuration that may be invoked from the base level, the composer has to offer a communication channel between the base level and the composed metaobjects.

This communication channel is based on the `perform` method. In a chain of composed metaobjects, when a metaobject receives a message, it checks if it knows how to handle it. If so, it performs the associated operation, otherwise it ignores it. In any case, it ends up forwarding this message to the next metaobject in the chain. Messages can be sent in order to invoke a particular method on a specific metaobject, or to perform a kind of *broadcast* within the composition chain. This mechanism is illustrated in section 4.3.

4 Reflex in practice

In this section we successively illustrate the use of Reflex for an architect of the metalevel through a MOP extension, for a metaobject programmer through the development of a metaobject, and for an application programmer through the creation and manipulation of a reflective object. We end by showing how the generic hook of Reflex can be used to control a new kind of event and support heterogeneity of metaobjects.

4.1 Perspective of the architect of the metalevel

Defining the MOP. The Reflex framework can be specialized in order to fit application requirements. The architect specializing the framework can introduce a new MOP, with a default implementation, define which hooks will be introduced and how. The lower part of Fig. 1 shows a specialization of the Reflex framework that we developed for applications for which trapping method invocations is a major feature. This specialization consists of definitions of:

- an extended MOP (interface `InvocationHandler`) that defines a method for handling method invocations, `handleInvoke`,
- a specific class builder (class `InvokeClassBuilder`) that introduces hooks for trapping invocations of public methods, through the new MOP method `handleInvoke`.

The hypothesis here is that performance is an issue with metaobjects performing, on average, very little work, which means that the cost of jumping to the metalevel should be minimized.

Defining a composition scheme. The architect of the metalevel can then design and implement a composition scheme. We already extended Reflex with such a scheme, inspired by Mulet et al. [16]. In this scheme (see Fig. 2) each *extension* metaobject performs its metaprocessing and then gives the control to the next metaobject in the cooperation chain. The chain composer ensures that there is always one and only one interpreter placed at the end of the chain. Therefore, extensions in this scheme are directly linked to another metaobject (which can be another extension or an interpreter) and explicitly cooperate with it.

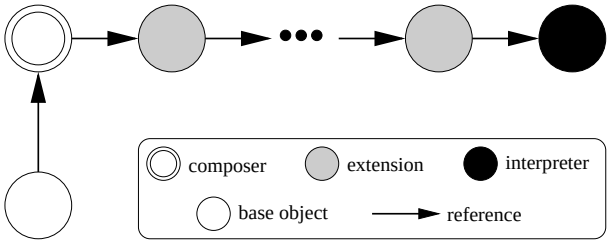

Fig. 2. Composition chain

4.2 Perspective of the metaobject programmer

Now that the metalevel architecture has been set up, the metaobject programmer can start implementing metaobjects. These metaobjects conform to the MOP defined by the architect of the metalevel, and possibly make use of a composition framework. We illustrate here the development of a configurable trace metaobject for the MOP defined by the `InvocationHandler` interface, using the composition framework presented above.

The `Trace` class is declared as implementing the base interface for extensions in the chain composition scheme. The role of such a metaobject is to trace, on a given output, the method invocations that occur on a set of base objects. This trace is selective: it applies to some methods only. To this end, the trace metaobject aggregates a hash set containing the names of the methods to trace. This hash set and the target print stream on which the trace is performed can be given at instantiation time or later, and can be updated dynamically. The simplified (without exception handling) implementation of the `handleInvoke` method is as follows (`out` is the target print stream object, and `toTrace` is the hash set containing the names of the methods to trace):

```
public Object handleInvoke(Method m, Object[] args,
                           ReflexObject receiver){
```

```
    if(toTrace.contains(methodname))                              (1)
        out.println("call: " + m.getName + " with: " + args );  (2)

    return this.getComposed().handleInvoke(m, receiver, args); (3)
}
```

First, the trace metaobject[4] checks if the hash set contains the name of the invoked method (1). If it does, a trace is produced, giving the name of the method and its arguments (2). The invocation is then forwarded to the next metaobject in the chain (3). Depending on the exact use of such a metaobject, there may be situations where a trace is generated for a fraction of the invocation, in which case, on average, the metaobject does not do much and performance becomes an issue. It makes sense to use a specific MOP.

Since the trace metaobject is a configurable one, different public services for setting it up are offered. These services include setting the methods to trace (`setMethodsToTrace(HashSet)`, `addMethodToTrace(String)`, etc.) and the print stream on which to perform the trace (`setOutput(PrintStream)`). All these public methods can seamlessly be invoked by the base level at any time, by using the `perform` method, as illustrated later on.

4.3 Perspective of the application programmer

Once the library of metaobjects up and ready, the application programmer can introduce metaobjects in his application. His role is basically that of identifying which objects should be reflective, and which metaobjects should be attached to each of these objects (even if this is actually hidden behind code transformation tools and wizards). Let us illustrate the creation of a reflective vector to which a trace metaobject is attached.

The first step is to create and set up the metaobject to be attached to the base object:

```
ChainComposer composer = new ChainComposer();  (1)
Trace trace = new Trace();                     (2)
composer.addExtension(trace);                  (3)
```

First, the composer is instantiated (1). Since no interpreter is specified in the constructor, the composer automatically instantiates a default interpreter, with default semantics Then the trace metaobject is created (2). Finally, the trace metaobject, which is an extension metaobject, is inserted into the composition chain managed by `composer` (3).

Once this is done, the reflective object can be instantiated, using the services of the `Reflex` class:

```
Vector v = (Vector) Reflex.createObject("java.util.Vector",
                                        composer);
```

[4] For a more realistic implementation of a trace metaobject, we refer the reader to the samples included in the Reflex package

The `createObject` arguments indicate that a reflective instance of `java.util.Vector` should be created, with the metaobject `composer` attached to the instance. When this statement is executed, the `createObject` method queries the composer for the class builder to use, and then delegates to the class builder the task of retrieving the reflective `Vector` subclass. At this point, if the reflective subclass has already been created, it is loaded if needed. Otherwise, it is created and loaded dynamically. The `createObject` method instantiates the reflective class, and attaches the metaobject to the created instance. Finally, it returns the reflective object.

Now, `v` is a reference of (declared) type `Vector` that points to a reflective object, instance of a reflective subclass of the `Vector` class. Let us suppose that, later on, the print stream used to perform the trace has to be updated. This must be done via an invocation of the `setOutput` method on the trace metaobject. Unfortunately, the metaobject is not directly accessible from the base-level object since the metalink only gives access to the composer. The composer being generic, it does not understand `setOutput`. This problem is circumvented by invoking the `perform` method on the composer. The first argument of the invocation is the `"setOutput"` string. The composer will forward this event to metaobjects of the composition chain. Eventually, the event will reach the trace metaobject, and will be processed:

```
PrintStream file = new PrintStream(
                       new FileOutputStream("trace.log"));  (1)
Reflex.sendEvent("setOutput", v, file);                     (2)
```

First, a print stream object is created on the desired output file (1). Then, a `"setOutput"` event is sent, together with its argument (2). This `sendEvent` method is static method of the `Reflex` class. This method is a useful shortcut which casts `v` to `ReflexObject`, retrieves a reference to the metaobject associated to it, and packs the arguments into an array of objects in order to be able to invoke the `perform` method.

4.4 Handling heterogeneous metaobjects

In this section, we illustrate how the generic hook of Reflex can be used to control a new kind of event and support heterogeneity of metaobjects.

In the context of mobile agents, an agent is normally seen as a closed entity which encapsulates all its data. Therefore, upon migration, all the objects that an agent references are passed by copy with it. However, this policy is not always desirable: some objects should be passed by reference, for instance if they belong to other agents or if they are huge and should be accessed lazily. This is feasible in Java using Remote Method Invocation (RMI) [22]. However, RMI is very demanding on the class to be passed by reference. It first has to implement the `Remote` interface, and, moreover, it has to be a subclass of `UnicastRemoteObject`. We would like a more flexible system allowing any object to be passed by reference, without any constraint on its class. Upon migration of

the agent, passing the object by reference means passing a proxy to the object rather than the object itself. This can be done by making the object reflective and attaching to it a particular metaobject. Such a metaobject is informed when serialization occurs and can specify an alternative object for serialization (the proxy). Once passed, the proxy is controlled by a metaobject which performs all method invocations remotely through the network.

We therefore need to give the control of serialization to metaobjects. The Java serialization API [21] offers a method, `writeReplace`, which can be used to specify an alternative object for serialization. Therefore, reflective objects should implement this method and a hook has to be inserted so that when this method is invoked automatically by the serialization process, the metalevel is informed and has the possibility of specifying an alternative object (such as a proxy).

We have implemented a new class of class builders, `SerializeClassBuilder`, which extends `InvokeClassBuilder`. The `SerializeClassBuilder` seamlessly adds a `writeReplace` method to each generated class using a `MethodCopier` object. This method, defined below, plays the role of a generic hook:

```
private Object writeReplace()
    throws ObjectStreamException {

 (1)   ReflexMetaobject mo = this.getMetaobject();
 (2)   Object replace = mo.perform("handleSerialize",
                                   null);
 (3)   if(replace == null)
 (4)       return this;
 (5)   return replace;
}
```

When serialization on the base object occurs, this method is invoked. It first retrieves the metaobject associated to the base object (1), and sends to this metaobject a `"handleSerialize"` event (2). If the result is null (3), i.e. no metaobject understood the operation to perform, then the default policy applies (4). Otherwise, the alternative object is returned (5).

If a metaobject wants to be informed when serialization occurs, it simply has to implement the `handleSerialize` method. Heterogeneous metaobjects can now cooperate, some understanding the `handleSerialize` method, some not. All existing metaobjects can be used seamlessly along with metaobjects newly developed for handling serialization.

5 Related work

The work on MetaJ [8] shares with ours the objective of making it possible to tailor reflective extensions to the specific constraints of its target applications. However, it only deals with meta-circular interpreters. On the one hand, this makes it possible to adopt a semantics-based approach, and therefore to be very

systematic and deal with formal correctness. On the other hand, the gap with practical considerations such as performance is far from being bridged.

On the opposite, all the practical Java reflective extensions [28, 26, 13, 18] provide a fixed MOP with some universal decisions made on the trade-offs between portability, expressiveness, and performance. The only point on which Reflex does not offer any freedom is portability, on the basis that this decision is actually set by the very definition of Java.

In spite of its fixed MOP, the case of Iguana/J [19] is peculiar. Indeed, its MOP seems to cover all the elementary execution events, for which metaobjects have already been designed. Iguana/J introduces then the idea of combining these metaobjects through protocol declarations, protocols which can then be, again declaratively, associated to base classes. This is a very elegant and flexible way of structuring customized metalevels from elementary building blocks with the protocols themselves providing higher-level building blocks. The implementation is based on the Java Native Interface and should therefore be still reasonably portable with an efficient capture of the basic events (but more implementation work). However, some points would require some clarifications. In particular, it seems that base classes are associated with protocols, which precludes the possibility of sharing metaobjects. Also, hook introduction is said to be performed by the native library! Finally, the model does not consider the possibility of performing hook introduction at the source code level or combining compile- and run-time MOPs. In Reflex, the introduction of class builders makes it much easier to combine these different approaches.

As for composing metaobjects, with the exception of MetaXa [13] and Guaraná[18], the other Java reflective extensions do not offer any help. In MetaXa, the composition scheme is fixed. The VM systematically organizes the metaobjects in a chain of metaobjects, following the order of introduction of these metaobjects. Guaraná is more open in that it offers metaobjects similar to the composers introduced in the specialization of Reflex as well as an extensible communication protocol similar to the propagation of events realized through the use of `perform`.

6 Future work

In its current state, Reflex still suffers from some limitations to claim to be a full-fledged open reflective extension of Java. We have already started working on these limitations.

6.1 Metalink cardinality

The current version of Reflex only supports a metalink cardinality 1-n, meaning that a base object can only be linked to one metaobject, and a metaobjet can control several base objects. Though a composition framework can be used to allow different metaobjects to participate in the control of a base object, a more mature open reflective extension should support a metalink cardinality n-n.

Supporting such a cardinality may enhance performance since, for a given object, different hooks can then directly give control to the relevant metaobject (or set of metaobjects). Metaobjects that are not interested by the corresponding event are not affected any longer.

6.2 Allowing more optimizations

Some applications do not require the flexibility provided by features such as the ability to dynamically change the set of metaobjects attached to a base object. In such a case, hooks should not be introduced, instead base and meta levels should be merged, like in compile-time MOP such as OpenJava [25].

Also, allowing the coexistence of non-reflective and reflective instances (possibly with different sets of controlled events) is a motivation for not directly modifying a base class and generating implicit subclasses. However, some applications do not require such a feature, e.g. when all instances of a class have to be reflective with the same set of controlled events. In this case, it is better to directly introduce hooks in the base class than to generate subclasses. This direct introduction of hooks into the base class should possibly be made at load time in order not to modify the standard version of the base class.

Conceptually, Reflex allows for such optimizations through the use of particular class builders. Some class builders could operate by merging the base and meta levels at compile time, some by introducing hooks statically in a class (or source) file, some others by introducing hooks in a base class at load time. However all these class builders have not been implemented yet.

6.3 Class builders and dynamic adaptability

A base-level object can be controlled by several metaobjects, either through the use of a composition framework in order to share a metalink, or through a metalink cardinality n-n. Each of these metaobjects is designed to react to particular kind of events, requiring the necessary hooks to be introduced in the class of the base-level object. As a consequence, the base-level object has to be an instance of a class that includes the union of all the hooks needed by the metaobjects used to control it.

Two issues derive from this statement. The first one deals with the specification of the required hooks and how they are mixed together in order to obtain the adequate class to instantiate. As of now, a metaobject has to be able to specify which class builder should be used. The composer then needs rules determining which available class builder meets the needs of all the composed metaobjects. We plan to make this specification finer-grained by allowing metaobjects to specify their needs in terms of basic transformations, and having a generic class builder able to compose all these transformations.

The second issue concerns the dynamic evolution of the set of metaobjects that control a base-level object. Conceptually, when adding a metaobject that requires hooks that are not present in the class of the base-level object, a new reflective class must be generated, with all the hooks. The instantiation link of

the base-level object has then to be changed in order to make it an instance of the newly generated class. However, changing the instantiation link is not possible in Java, nor is it possible to really *replace* an existing object by a new one. The only implementable solution we envision at this time is that of obtaining a shallow copy, instance of the new class, similarly to what is done when "converting" a non-reflective object to a reflective one. However, this solution is not satisfactory for obvious identity problems. The possibility of class reload, offered by the upcoming JDK 1.4 [1] may help solve this issue.

6.4 Performance

One of the goals of Reflex is to minimize performance loss by providing the minimum reflective system needed for a particular application. In order to validate the gain, we plan to carry performance tests between Reflex and other non-customizable reflective extensions. Also, we plan to compare the use of the generic MOP of Reflex to that of a specialized MOP. Beforehand, we plan to study how the implementation of the base components of Reflex could be optimized.

7 Conclusion

This paper has presented Reflex, a prototype *open* reflective extension of Java. As such, Reflex is a specializable framework and a toolkit that can be used to build or adapt reflective extensions that meet particular needs. It fills the gap between low-level byte-code manipulation APIs allowing the definition of custom-built reflective extensions and high-level non-specializable reflective extensions through two main concepts: a generic MOP and class builders.

As of now, a working specialization of Reflex has been implemented and applied to mobile object systems. We plan to pursue the validation of Reflex with new applications, testing different composition schemes, and work on the limitations to Reflex openness.

The Reflex package (binaries, source code, samples and documentation) can be obtained from the web at the following URL:

`http://www.dcc.uchile.cl/~etanter/Reflex`

Acknowledgements

This research is supported in part by the EU-funded IST Project 1999-14191 EasyComp.

References

1. V. Aggarwal. The magic of Merlin – how the new JDK 1.4 levitates its functionality. *Java World*, March 2001.

2. M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Boston, MA, USA, January 2000. ACM Press. ACM SIGPLAN Notices, 34(11).
3. J.-P. Briot, R. Guerraoui, and K.-P. Lhr. Concurrency and distribution in object oriented programming. *ACM Computer Surveys*, 30(3), September 1998.
4. D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. In *Concurrency Practice and Experience*, volume 10. Wiley & Sons, September 1998.
5. S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *ECOOP 2000 - Object-Oriented Programming - 14th European Conference*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.
6. S. Chiba and M. Tatsubori. Yet another java.lang.class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.
7. P. Cointe, editor. *Proceedings of Reflection '99*, volume 1616 of *Lecture Notes in Computer Science*, Saint-Malo, France, 1999. Springer-Verlag.
8. R. Douence and M. Südholt. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation*, 14(1), 2001. To appear.
9. S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Objet-Oriented Programming*, June:39–50, 1999.
10. *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, Boston, MA, USA, January 2000. ACM Press. ACM SIGPLAN Notices, 35(7).
11. B. Foote and R.E. Johnson. Reflective facilities in Smalltalk-80. In N. Meyrowitz, editor, *OOPSLA'89, Conference Proceedings*, pages 327–335, New Orleans, Louisiana, USA, October 1989. ACM SIGPLAN Notices, 24(10).
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
13. M. Golm and J. Kleinöder. Jumping to the meta level, behavioral reflection can be fast and flexible. In Cointe [7], pages 22–39.
14. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.
15. SUN Microsystems. Dynamic proxy classes. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`, 1999.
16. P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA'95*, pages 316–330. ACM Press, October 1995.
17. H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In *Proceedings of ECOOP'94*, pages 299–319, 1994.
18. A. Oliva and L. E. Buzato. Composition of meta-objects in Guarana. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems (COOTS'99)*, San Diego, California, USA, May 1999.
19. 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.
20. R.J. Stroud and Z. Wu. Using metaobject protocols to satisfy non-functional requirements. In C. Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, pages 31–52. CRC Press, 1996.

21. SUN Microsystems. *Object Serialization*, 1998. `http://java.sun.com/products/jdk/1.2/docs/guide/serialization/`.
22. Java Remote Method Invocation specification. Technical report, SUN Microsystems, 1999. `http://java.sun.com.products/jdk/1.2/docs/guide/rmi/`.
23. E. Tanter. Reflex, a reflective system for Java — application to flexible resource management in Java mobile object systems. Master's thesis, Universidad de Chile, Chile – Vrije Universiteit Brussel, Belgium, 2000.
24. E. Tanter. *Reflex Website*, 2001. `http://www.dcc.uchile.cl/~etanter/Reflex`.
25. M. Tatsubori. An extension mechanism for the Java language. Master's thesis, University of Tsukuba, Japan, 1999.
26. I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Cointe [7], pages 2–21.
27. 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++*, October 1998.
28. Z. Wu and S. Schwiderski. Reflective Java: Making Java even more flexible. APM 1936.02, APM Limited, Castle Park, Cambridge, UK, February 1997.