Mirages: Behavioral Intercession in a Mirror-based Architecture

Stijn Mostinckx[1*] Tom Van Cutsem[1†] Stijn Timbermont[1*] Éric Tanter[2‡]

[1]Programming Technology Lab
Vrije Universiteit Brussel, Belgium

[2]CWR/Computer Science Dept
University of Chile, Chile

{smostinc,tvcutsem,stimberm}@vub.ac.be etanter@dcc.uchile.cl

ABSTRACT

Mirror-based systems are object-oriented reflective architectures built around a set of design principles that lead to reflective APIs which foster a high degree of reusability, loose coupling with base-level objects and whose structure and design corresponds to the system being mirrored. However, support for *behavioral intercession* has been limited in contemporary mirror-based architectures, in spite of its many interesting applications. This is due to the fact that mirror-based architectures only support explicit reflection, while behavioral intercession requires implicit reflection. This work reconciles mirrors with behavioral intercession. We discuss the design of a mirror-based architecture with implicit mirrors that can be absorbed in the interpreter, and *mirages*, base objects whose semantics are defined by implicit mirrors. We describe and illustrate the integration of this reflective architecture for the distributed object-oriented programming language AmbientTalk.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages

Keywords

Reflection, Metaprogramming, Mirrors, Mirages, AmbientTalk

1. INTRODUCTION

Computational reflection [23, 17] provides programs with a well-defined interface to reason about themselves. Reflection is often further refined according to what kind of reasoning is allowed and what parts of the program can be reasoned about. A reflective architecture supports *structural introspection* if it allows programs to inspect the structural aspects of a program. It allows for *structural intercession* if programs can modify their structure. It supports *behavioral introspection* if programs can inspect their runtime behavior (e.g. a stack trace). It allows for *behavioral intercession* if programs can change their behavior, using custom metaobjects to change the semantics of the language itself. Computational reflection has been widely adopted in object-oriented languages (e.g. Java, Self, Smalltalk, CLOS), although they differ greatly in terms of the reflective power they convey.

In this paper, we present the metaobject protocol of AmbientTalk, a distributed actor-based object-oriented language. In previous work, we have explicitly presented AmbientTalk as a "language laboratory" for experimenting with novel language constructs in the context of volatile, ad hoc networks [9]. More concretely, we realized this "language laboratory" by making AmbientTalk a reflective language, such that novel language constructs can be expressed within the language itself. Whereas our previous metalevel architecture provided adequate support for behavioral intercession, it lacked a modular, stratified design.

Bracha and Ungar have proposed a set of design principles for the design of a *mirror-based* metaobject protocol: a reflective API which fosters a high degree of reusability, loose coupling with base-level objects and whose structure and design directly corresponds to the system being mirrored [5]. Therefore, mostly influenced by Self's mirrors [1], we decided to redesign the AmbientTalk architecture in a mirror-based way. While mirror-based architectures usually provide proper access to the structure of programs, their support for behavioral intercession has been relatively limited. However, behavioral intercession is a key enabler for the reflective implementation of language constructs.

This paper reports on our experiments in reconciling AmbientTalk's mirror-based architecture with behavioral intercession. We introduce the *mirage*: a base-level object whose semantics is described by a custom *implicit mirror*, i.e. it is an object with a custom metaobject protocol. We describe the design issues that arise from introducing mirages in a mirror-based architecture, describe the introduction of futures – a distributed language construct – as a use case of mirages and show how mirages are implemented with moderate effect on the overall performance of the system.

Availability An AmbientTalk interpreter with explicit support for behavioral intercession through mirages is available at `http://`

Draft version. Revised version accepted at DLS'07, October 22, 2007, Montréal, Québec, Canada.

*Author funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

†Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

‡É. Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

prog.vub.ac.be/amop. The included standard library contains the complete code for the futures language construct outlined in this paper.

2. MIRROR-BASED REFLECTION

Bracha and Ungar define a mirror-based architecture as any reflective architecture that adheres to three key design principles, to wit *encapsulation*, *stratification* and *ontological correspondence*[5]. In what follows, we describe what is meant by each of these principles in the context of a mirror-based reflective architecture.

2.1 Encapsulation

The principle of encapsulation states that metalevel entities should *encapsulate* their implementation details [5]. In essence, it should be possible to write metalevel programs (source code browsers, pretty-printers, debuggers, object inspectors) against an abstract API, which fosters a higher degree of reuse because the API can serve as an abstraction barrier for multiple implementations. For example, consider that we want to reuse as much code as possible from existing metaprograms to be able to debug or inspect objects on a *remote* virtual machine. When the metaprograms only code against an interface, rather than a specific reflective implementation, large parts of the code can be reused without change.

To enable metalevel entities to encapsulate their implementation, a necessary (but not necessarily sufficient) condition is that their type should expose only their interface, not their implementation. This rules out nominal type systems based on classes (implementation), as e.g. employed by Java or C++. The Java reflection API, for example, ties metalevel representations to a specific implementation, inhibiting reuse. On the other hand, the Java Debugger Interface is a reflective API based on *interface* types. Hence, clients are shielded from specific implementation classes [5]. Structural type systems (as e.g. employed by StrongTalk [4]) or dynamically typed languages inherently avoid such encapsulation breaches.

Of course, using e.g. a dynamically typed language does not necessarily imply that a reflective API preserves encapsulation. For example, if it is desirable that a reflective API can be used on both local and remote objects without substantial changes in the client, the API still has to be designed accordingly.

2.2 Stratification

The principle of stratification states that metalevel entities should be cleanly separated from base-level functionality [5]. This separation ensures among others that when a base-level method's name corresponds to a metalevel operation, this method is not accidentally regarded as part of metaobject protocol.

A stratified design also implies less coupling between the base- and metalevels. Reduced coupling has benefits in terms of *deployment*: if access to the metalevel architecture can be easily trapped, it is easier to deploy programs *without* reflective support if it can be derived that programs never access it, or at least to *postpone* the activation of reflective support until it is required by the application.

The principles of encapsulation and stratification are also innately connected. In order for reflection to be stratified, base-level objects should not contain any explicit reference to metalevel entities. The very presence of such a link often breaks encapsulation and stratification. For example, invoking `obj.getClass()` on a Java object links the object directly to its metalevel representation. This makes it hard for metalevel programs to uphold encapsulation. For example, if `obj` is an instance of a proxy class, perhaps a metalevel program would like to hide this fact from its metalevel clients. This is virtually impossible given the hard-wired link from the base- to the metalevel.

Another example of a violation of stratification occurs in Smalltalk. Performing `obj class` results in a reference to the class of an object. In Smalltalk, classes play a dual role: they are used both for base-level tasks such as instance creation (e.g. `self class new`) and for metalevel tasks such as code browsing (e.g. `obj class subclasses`). Because of this dual role, it becomes hard to deploy Smalltalk applications without the reflective capabilities of classes.

In a mirror-based architecture, access to the metalevel should be a dedicated, explicit operation, such that it is not normally used by regular base-level programs. Moreover, when metalevel programs can intervene in the execution of this operation, they can preserve the encapsulation of the metalevel representation of base-level objects. For example, in StrongTalk the reflective API can only be accessed by performing `Mirror on: obj` [5]. Likewise, in Self a mirror on an object is created by performing `reflect: obj` [1]. These methods often serve as factory methods for the creation of appropriate mirrors on objects. The downside is that access to the metalevel is not a polymorphic message send, such that methods like `reflect:` often have to perform some internal dispatching based on the object's type.

2.3 Ontological Correspondence

The principle of ontological correspondence states that the metalevel should be structured according to the same concepts and rules that govern the base-level [5]. Bracha and Ungar further distinguish between *structural* and *temporal* correspondence, which corresponds to the distinction between *code* (a description of a computational process) and *computation* (the actual execution of that process).

A mirror-based architecture that is temporally correspondent should make the distinction between code and computation manifest in its API. The advantage is that the API that reflects on code can be used both for reasoning about pure source code, as well as for reasoning about code that has been turned into live objects. For example, when writing a code browser against such an API, it becomes easy to use the browser both for viewing code loaded from a database, as well as for inspecting live objects or perhaps even serialized objects.

Structural correspondence implies that every language construct has a reified representation at the metalevel [5]. In a truly structurally correspondent mirror-based architecture, this principle requires that even the body of a method should have a metalevel representation. However, reasoning about the body of a method brings us on dangerous grounds. If the method has been compiled into e.g. bytecode, it does not suffice to provide a representation for bytecodes in the reflective API: the bytecodes are concepts from a different language, i.e. the virtual machine language. If exposed directly to the reflective API of the high-level language, transformations employed by the compiler may present clients of the reflective API with inconsistent information. Hence, a structurally-correspondent mirror architecture ideally provides separate APIs for reasoning about each distinct language in the system [5].

2.4 Summary

An ideal mirror-based system:

- provides a reflective API based on interfaces which preserves the encapsulation of metalevel objects.
- factors the link from base-level objects to metalevel objects out of the base-level objects themselves. This stratifies base- and metalevels, making it easier for metaprograms to preserve encapsulation, and making it easier to disable reflection when it is not required.
- makes the distinction between APIs that manipulate code and those that manipulate computation manifest. The API that reflects on code does not require a running computation to reflect upon.
- reifies every element of the base-level language. Language features that are transformed, optimized or desugared should remain intact when mirrored by the language's reflective API.

3. THE AMBIENTTALK LANGUAGE

Having presented the design principles underlying a mirror-based architecture, we present a concrete embodiment of these principles in the reflective architecture of AmbientTalk, a distributed object-oriented programming language. The language described here is actually AmbientTalk/2, an updated version of the language whose reflective API differs from the version presented in previous work [9]. In the remainder of this paper, we will simply refer to the updated language as AmbientTalk.

This section begins with a bird's-eye overview of AmbientTalk's object model. Subsequently, we describe the introspective mirror infrastructure which allows reflecting on standard objects. Section 3.3 subsequently introduces AmbientTalk's actor-based concurrency model and illustrates how the actor mirror can be used to group reflective behavior shared between all objects belonging to the same actor. Finally, we demonstrate how the metalevel architecture conforms to the criteria outlined in the previous section.

3.1 Base-level objects

AmbientTalk is an object-based language. Objects are not instantiated from classes. Rather, they are either created *ex-nihilo* or by cloning and adapting existing objects. AmbientTalk objects consist of field and method slots. Consider the definition of a prototypical planar point object:

```
// Point is a prototypical point object
def Point := object: {
  // define two fields
  def x := 0;
  def y := 0;
  // definition of methods
  // this method serves as the "constructor"
  def init(newx, newy) {
    x := newx;
    y := newy;
  };
  def +(other) {
    self.new(x+other.x, y+other.y)
  };
  def distanceToOrigin() {
    (x*x + y*y).sqrt();
  };
}
```

The above code defines a new anonymous object and binds it to a variable named `Point`. This object serves as a prototypical point object and can be used to create clones:

```
def p := Point.new(1,2);
```

Every object understands the message `new`, which creates a clone of the receiver object and initializes the clone by invoking its `init` method with the arguments that were passed to `new`. This protocol closely corresponds to that of class instantiation, but rather than allocating a new empty object from a class, a clone is created from a prototype.

By convention, when an object receives a message it does not understand, it *delegates* the message to the object bound to its slot named `super`. We employ the delegation semantics of Self [26] and Act1 [15]: a delegated message is a message that is forwarded to another object, but the `self` pseudo-variable remains bound to the delegating object. Hence, AmbientTalk supports object-based (single) inheritance. The `super` slot is assignable, such that the "parent" of an object may change. This enables *dynamic inheritance* which is useful for implementing objects that can change states [26]. A declarative syntax is provided for specifying that a new object delegates to an existing prototype:

```
// the SpatialPoint prototype delegates to Point
def SpatialPoint := extend: Point with: {
  def z := 0;
  ...
}
```

In the above example, `SpatialPoint` and `Point` remain separate objects in their own right. The `extends` relationship between a child and a parent object implies that the child's `super` field is initialized to the parent object and that when a child is cloned, the clone's `super` field is bound to a clone of the parent object. Hence, when a `SpatialPoint` is cloned, the clone has its own `Point` parent object with its own copies of the `x` and `y` fields.

AmbientTalk provides support for block closures reminiscent of those in Self and Smalltalk. A block closure is an anonymous function object that encapsulates a piece of code and the bindings of lexically free variables and `self`. Block closures are constructed by means of the syntax `{ |args| body }`, where the arguments can be omitted if the block takes no arguments. The following code excerpt shows a typical usage of blocks to remove all elements from a collection that fail to satisfy a predicate:

```
def from: collection retain: predicate {
  result := clone: collection; // shallow copy
  collection.each: { |elt|
    predicate(elt).ifFalse: {
      result.remove(elt)
    }
  };
  result;
};
from: [1,-2,3] retain: { |e| e > 0 }
```

Note that AmbientTalk supports both traditional canonical syntax (e.g. `o.m(a,b,c)`) as well as keyworded syntax (e.g. `dict.at:`

k put: v) for method definitions and message sends. As a general rule, we use keyworded syntax for control structures (e.g. while:do:) or language constructs (e.g. object:). The canonical syntax is used for expressing application-level behavior.

3.2 Introspective Mirrors

AmbientTalk has a mirror-based architecture that has been inspired by that of Self [1]. The following code excerpt gives some example uses of introspecting objects by means of their mirror:

```
// retrieve a mirror by invoking reflect:
def mirrorOnP := (reflect: p);
// read the contents of a field via its mirror
mirrorOnP.grabField(`x).value; // 1
// retrieve a mirror on a method
mirrorOnP.grabMethod(`init); // <mirror on method:init>
// reflectively invoke a method
mirrorOnP.invoke(Message.new(p, `distanceToOrigin, []));
// print all method names
mirrorOnP.listMethods().each: { |method|
  system.println(method.name)
};
// add a z coordinate
mirrorOnP.addField(Field.new(`z, 0));
```

As can be seen from the examples, mirrors support introspection (retrieval of field and method mirrors), invocation (explicit invocation of methods) and self-modification (addition of fields and methods). The Message object passed to invoke encapsulates a receiver (any object), a selector (a symbol) and actual arguments (an array). The receiver parameter is the object to which self is bound during method invocation.

Mirrors on objects are created by means of the reflect: construct. This ensures that the creation of the appropriate kind of mirror is separated from any base-level concerns. The reflect: construct creates a mirror by calling a factory method, which can be replaced by metaprograms. This is explained in more detail in the following section.

3.3 Mirrors on Actors

AmbientTalk is a concurrent actor-based [2] language. While we will not go into the details of AmbientTalk's concurrency features, we have to briefly describe actors in order to give a complete view of the mirror architecture. AmbientTalk does not represent objects as active objects. Rather, it adopts the communicating event loops model of the E programming language [19], in which an actor is conceived as an event loop which contains regular objects, shielding them from harmful concurrent modifications. Each regular object is said to be *owned* by exactly one actor. Only the owning actor of an object may execute its methods.

Objects owned by one actor can only communicate with objects owned by another actor by means of asynchronous message passing: a message sent to an object owned by another actor is enqueued in the owner's message queue and processed by the owner itself at a later point in time. AmbientTalk borrows from E the syntactic distinction between synchronous sends (e.g. o.m()) and asynchronous sends (expressed as o<-m()). The beneficial concurrency properties of this event loop architecture can be found elsewhere [19].

Each actor hosts both base-level objects (representing an application) and metalevel objects (mirroring base objects). Each actor also hosts an *actor mirror*, a special object denoting the mirror on the actor as a whole. This mirror is special in that it does not reflect upon a concrete base-level object because an AmbientTalk actor is an event loop rather than a concrete object. The actor mirror allows manipulating the event loop without exposing its implementation, just like a java.lang.Thread object in Java allows for the manipulation of a thread without exposing its implementation. The actor mirror also hosts metalevel behavior which is shared by all of the objects it owns. The operations reified by the actor mirror are those which transcend the scope of a single object (e.g. the creation and sending of asynchronous messages to communicate with remote objects).

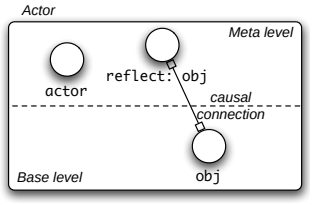

Figure 1: Layout of an AmbientTalk actor.

Figure 1 gives an overview of the different objects that constitute an actor. The actor mirror is bound to the actor field in the global scope. An actor mirror can be accessed without passing via the mirror factory. This does not violate stratification because actor is already a pure metalevel entity.

The mirror factory method is defined in the actor mirror. reflect: obj is implemented as actor.createMirror(obj). Metalevel programmers may install a custom actor mirror at runtime. By overriding createMirror in the custom actor mirror, it becomes possible to customize the mirrors of all objects owned by the actor. As an example, consider the following code excerpt which installs a custom actor mirror that overrides only the mirror factory method. The custom factory returns "sealed object" mirrors which disallow the explicit addition of fields to an object at the metalevel.

```
actor.install: (extend: actor with: {
  def createMirror(onObj) {
    extend: super.createMirror(onObj) with: {
      def addField(field) {
        raise: IllegalOperation.new(
          "Sealed object: field addition prohibited.");
      }
    }
  }
})
```

After the installation of a custom actor mirror, actor is bound to the extended mirror, such that all calls to reflect: within the same actor use the new mirror factory. Section 5.3 presents an additional example where a custom actor mirror is installed to hook into the asynchronous message sending protocol of the actor.

3.4 Evaluation

In this section, we briefly describe why AmbientTalk's metalevel architecture can be regarded as a mirror-based architecture, by showing how it exhibits the three properties described in section 2.

AmbientTalk mirrors preserve encapsulation. This is primarily because AmbientTalk is a dynamically typed language. Hence, any

object can be returned from a call to `reflect:` as long as it implements the metaobject protocol appropriately.

AmbientTalk's mirror architecture is stratified: mirrors are not accessed from the base object they reflect, but rather need to pass via a *mirror factory* which can be customized by metaprograms. Similarly, the actor mirror is stratified, since it contains only metalevel behavior.

AmbientTalk's mirror architecture is structurally correspondent to the base-level: mirrors reflect all operations applicable on objects. Also, because AmbientTalk uniformly represents all base-level entities (e.g. numbers, block closures, parse trees) as objects, every element of the language can be mirrored. The issue of requiring a separate API for high-level and low-level language does not apply to AmbientTalk: the interpreter currently uses the parse trees themselves to evaluate method bodies, hence there is no low-level language to reflect upon.

AmbientTalk's mirror architecture is not temporally correspondent: mirrors do not explicitly distinguish code from computation. It is not possible to introspect on the source code of an object using the same interface to introspect on the object itself.

4. MIRAGES: INTERCESSIVE MIRROR-BASED REFLECTION

Behavioral intercession has traditionally been introduced in languages to allow programs to modify parts of their own semantics [23, 17]. As such, it has a huge number of applications. In particular, it can be used as a general framework to introduce new data types in a programming language such as proxy objects (which trap invocations and forward them to their principal), persistent objects (which trap slot assignments and update the persistent storage accordingly), and so on [29]. As a language laboratory, AmbientTalk relies on behavioral intercession to develop new language constructs for mobile ad hoc networks [9].

Behavioral intercession requires a different kind of reflection from that provided by mirror-based architectures such as the one described in the previous section, or that of Self and Strongtalk. These architectures allow for *explicit* reflection, that is, metacomputation is triggered explicitly by programs using mirrors, whereas behavioral intercession requires *implicit* reflection, where metacomputation is triggered implicitly by the interpreter as a result of evaluating base code [18]. In the following, we first illustrate this issue of implicit reflection, and introduce a distinction between explicit and implicit mirrors. Subsequently, in Section 4.2, we describe how implicit mirrors can be absorbed by the interpreter by means of dedicated *mirage* objects, thereby enabling behavioral intercession in a mirror-based architecture.

4.1 Explicit vs. Implicit Mirrors

To illustrate the difference between explicit and implicit reflection, and pinpoint what is lacking in mirror-based architectures, consider the implementation of a simple metaprogram that logs all methods invoked on an object. Because mirrors support the `invoke` operation, a metalevel programmer can install a custom mirror factory returning mirrors that override the `invoke` method as follows:

```
actor.install: (extend: actor with: {
  def createMirror(onObj) {
    extend: super.createMirror(onObj) with: {
      def invoke(invocation) {
        system.println("invoked "+invocation.selector);
        super.invoke(invocation); // default behavior
      }
    }
  }
})
```

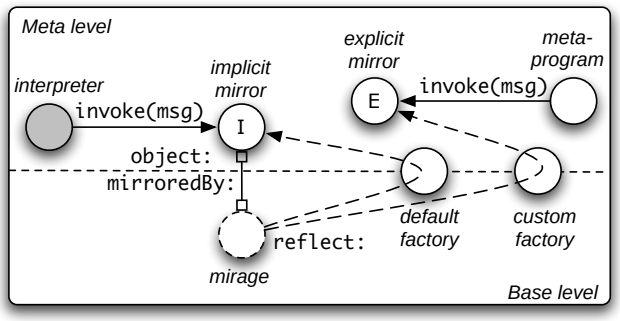

Figure 2: Implicit versus Explicit Mirrors

However, the result of installing this mirror is that only invocations performed *explicitly* upon the mirror are logged (e.g. `(reflect: o).invoke(invocation)`). When the interpreter is evaluating a standard base-level invocation on the referent of that mirror (e.g. `o.m()`), no logging happens. This is because the interpreter uses an implicit implementation of the `invoke` operation, rather than consulting the mirror provided by the mirror factory. In other words, the logging mirror is not *absorbed* by the interpreter.

One approach to introduce implicit reflection, and hence behavioral intercession, in a mirror-based architecture would be to make the interpreter consult the mirror factory rather than using implicit implementations for metalevel operations. However, this approach is impractical for a number of reasons. Perhaps the most obvious one is related to performance: having the interpreter consult the mirror factory for *every* meta-operation on *every* object would impose an unacceptable performance penalty on the application. But more importantly, having the interpreter absorb mirrors confuses two fundamentally different kinds of reflection and could simply break the interpretation of objects. Consider the sealed object mirror introduced in section 3.3: it can be used to ensure read-only reflection by metaprograms such as object inspectors; however if it is absorbed by the interpreter, the interpreter *itself* would be precluded from adding slots to an object, making it impossible to instantiate base-level objects.

Our solution to this dilemma is to introduce two kinds of mirrors: *i) explicit mirrors*, for use by metaprograms, such as the sealed object mirror; *ii) implicit mirrors*, to be absorbed by the interpreter, enabling behavioral intercession, such as the log mirror. The difference between both kinds of mirrors is illustrated in figure 2. The figure shows a mirage causally connected to its implicit mirror. The implicit mirror is absorbed such that when the interpreter manipulates the mirage, it uses its implicit mirror without consulting the mirror factory. Metalevel programs on the other hand need to pass via the mirror factory which *may* return the implicit mirror (as is done by the default mirror factory), but can also return another explicit mirror, such as the sealed object mirror described previously.

Implicit and explicit mirrors can be distinguished according to the following characteristics:

Reflection Type The fundamental distinction between explicit and implicit mirrors is the type of reflection they enable: explicit mirrors enable explicit reflection, while implicit mirrors enable implicit reflection.

Cardinality Since an implicit mirror is effectively absorbed by the interpreter and henceforth used in the actual interpretation process of that object, there is a strict one-to-one correspondence between an object and its *unique* implicit mirror[1]. Conversely, objects can be reflected upon by *multiple* and *unrelated* explicit mirrors, each providing a different form of reflective access to its referent. For instance, when reflecting upon a proxy object for a remote object, two explicit mirrors can be conceived: one which reifies the proxy object itself and one which reifies the remote object.

Completeness Unlike explicit mirrors whose interface is only constrained by their use in the program, implicit mirrors are required to provide a *complete* implementation of the metaobject protocol. This is a direct consequence of the fact that implicit mirrors are absorbed by the interpreter which can invoke any method of the metaobject protocol.

Finally, note that an implicit mirror can be seamlessly used as an explicit mirror. For example, the default mirror factory returns the implicit mirror of an object as its default explicit mirror. As previously mentioned, the opposite relation does not necessarily hold because an explicit mirror is not necessarily complete and may impose restrictions that can break the interpreter.

4.2 Absorbing Mirrors using Mirages

Mirror-based architectures provide means to define new explicit mirrors on objects by hooking into the mirror factory, yet they lack the notion of an implicit mirror. To avoid having to absorb explicit mirrors, implicit mirrors are not created by means of a mirror factory but rather introduced using the concept of a *mirage*. A mirage is an "immaterial" object whose semantics is entirely described by an implicit mirror. A mirage behaves as a regular object, but consists of a special base-level object causally connected to an implicit mirror that defines its MOP.

The causal connection between a mirage and its implicit mirror is established in two steps. First, a prototype object must be created, which will serve as the mirror object, defining the semantics of the object it mirrors. Then, copies of that prototype can be used as the implicit mirror of new mirages.

4.2.1 Mirror Prototypes

Any object can serve as an implicit mirror for a mirage as long as it provides a *complete* implementation of the AmbientTalk metaobject protocol. To facilitate the development of mirror objects which require only small changes with respect to the default language semantics, the actor mirror contains a prototypical mirror object named the `defaultMirror` which encapsulates AmbientTalk's default metaobject protocol. The `defaultMirror` makes the native metaobject protocol implementation explicitly accessible while keeping it encapsulated behind the MOP interface. Most implicit mirrors extend the default mirror to implement their custom semantics.

Reconsider the logging example from the previous section. In order to log all messages sent to an object, it is necessary to first define a prototypical logging mirror object which redefines the `invoke` metalevel operation:

```
def LogMirror := extend: actor.defaultMirror with: {
  // override invoke to log the message
  def invoke(invocation) {
    system.println("invoked "+invocation.selector
      +" on "+self.base);
    super.invoke(invocation); // default behavior
  };
}
```

The above `LogMirror` serves as a prototypical logging mirror. It has not been tied to a mirage yet, and hence has not yet been absorbed by the interpreter. The `defaultMirror` it extends similarly is such a prototype mirror. Note that these prototype mirrors are *not* causally connected to any object at this point. To be absorbed, a prototype mirror must be associated to a mirage.

4.2.2 Mirage Creation

A mirror object can only be absorbed by the interpreter when a mirage object is defined to be explicitly mirrored by that mirror object. The code excerpt below redefines the `Point` prototype from section 3 as a mirage, whose behavior is now defined by the `LogMirror` presented in the previous section.

```
def Point := object: {
  def x := 0;
  def y := 0;
  def init(newx, newy) { ... };
  def +(other) { ... };
  def distanceToOrigin() { ... };
} mirroredBy: LogMirror
```

The `object:mirroredBy:` language construct first creates a new, empty mirage object. The empty mirage needs to be associated with an implicit mirror which describes its semantics. The required implicit mirror is created by copying the specified mirror object, passing the empty mirage to the constructor of the new mirror. The mirror is then set as the implicit mirror of the empty mirage. From that point on, the mirage and its mirror are causally connected and the new instance of the mirror is effectively absorbed by the interpreter. This is illustrated in Figure 3. Only after the mirror has been absorbed is the initialization code of the object definition executed, such that this code is properly reflected by the new implicit mirror. For example, the field definitions for `x` and `y` are reified as `addField` invocations on the `LogMirror`.

In section 3.1, it was explained that when an object extends another object, the parent object is cloned when the child object is cloned. Because the `LogMirror` extends the `defaultMirror`, the `defaultMirror` is also instantiated when the `LogMirror` is used to create a new mirage. The constructor of the `defaultMirror` initializes its `base` field to refer to the new, empty mirage

[1]Of course, the implicit mirror bound to a base-object can be the result of a *composition* of multiple implicit mirrors, however this composition needs to be semantically coherent [20].

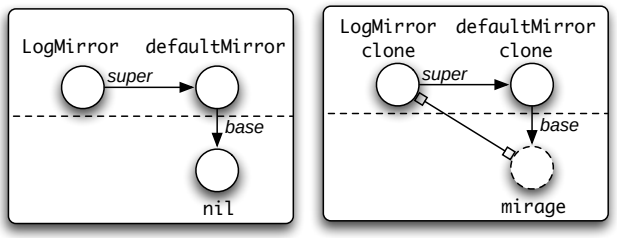

Figure 3: Left: an unabsorbed mirror prototype. Right: a mirage causally connected to an absorbed instance of the mirror prototype.

object. This ensures that when an absorbed `LogMirror` instance invokes `self.base` while logging an invocation, this field will refer to a causally connected `Point` mirage.

To base-level code, the logged `Point` mirage behaves like any other AmbientTalk object. This mirage may be instantiated or cloned. The default cloning and instantiation semantics (that can be overridden at the metalevel) uphold the one-to-one correspondence between the mirage and its implicit mirror. When a mirage is cloned, its implicit mirror is cloned and vice versa. Hence, clones are always created in pairs such that they too can become causally connected.

4.2.3 Summary

AmbientTalk introduces support for behavioral intercession in mirror-based architectures by distinguishing *implicit* mirrors from *explicit* mirrors. Unlike explicit mirrors, implicit mirrors are not defined by adapting a mirror factory. Rather, they are *absorbed* by the interpreter when a new mirage object is created by means of the `object:mirroredBy:` language construct.

4.3 Mirages and Stratification

The introduction of mirages in a mirror-based architecture may at first sight jeopardize its adherence to the design principles advocated by Bracha and Ungar [5]. The encapsulation principle is upheld: an implicit mirror properly encapsulates the metalevel behavior of the mirage and a mirage need not be aware of the implementation details of its mirror. The stratification principle is upheld even though there exists a one-to-one correspondence between mirages and their implicit mirror. Although the metalevel mirror object must be explicitly tied to the base-level mirage object, base- and metalevel code remain strictly separated in different objects. One advantage of this strict separation is that base-level methods cannot accidentally override metaobject protocol methods and vice versa.

Whether or not an object is a mirage is not leaked to other base-level code. Once a mirage is created, it is indistinguishable from an ordinary object. Since mirages are treated identical to ordinary objects, the only way to reflect upon them is by using the `reflect:` construct. Since this ensures that the mirror factory is consulted, a custom explicit mirror can be returned. For instance, when reflecting upon a `Point` mirage (as defined in the previous section), the returned explicit mirror may be the sealed object mirror presented in section 3.3. This illustrates that mirages enjoy the same loose coupling with their explicit mirrors as any other object.

As noted in section 2.2, the stratification principle facilitates the deployment of base-level programs separate from the deployment of reflection support. In spite of the fact that reflective access to implicit mirrors is stratified, the use of mirages does necessitate the presence of reflective infrastructure. With respect to deployment, code that uses the `object:mirroredBy:` construct must be regarded similar to code that uses the `reflect:` construct.

5. MIRAGES APPLIED: FUTURES

In this section, we demonstrate the use of behavioral intercession by means of a concrete language construct, namely future-type message passing [28]. Future-type message passing is a classic technique to allow asynchronous messages to return a result, without resorting to explicit callback messages. We first describe the base-level behavior of futures in AmbientTalk. Subsequently, we describe the role of behavioral intercession in the reflective implementation of futures. Finally, we show how to integrate futures with the asynchronous message passing protocol of AmbientTalk actors.

5.1 Future-type Message Passing

By default, an asynchronous message send has no return value (i.e. it returns `nil`), forcing the programmer to rely on manual callback methods to obtain the result of an asynchronous computation. Future-type message passing reconciles asynchronous message sends with return values, by making an asynchronous send immediately return a *future* object [28, 16]. A future is a placeholder object (i.e. a proxy) which is eventually *resolved* with the return value. The code excerpt below illustrates future-type message passing in AmbientTalk.

```
def database := DBManager<-connect("myDB","user","pass");
def employees := database<-query("SELECT * FROM Employee");

when: employees becomes: { |table|
  system.println(table)
}
```

In the above example an asynchronous message is sent to create a connection to a database. The resulting future object is stored in the `database` variable. Subsequently, an asynchronous `query` message is sent to the `database` future, which buffers the message and forwards it to its resolved value once this value is available. Note that only asynchronous messages can be sent to a future object. This ensures that the message can be delayed by the future as long as the return value is not yet available.

In traditional approaches, when code requires synchronous access to the actual return value of an asynchronous send, the thread executing the code is suspended until the future is resolved [7]. However, because AmbientTalk actors are event-driven (as explained in section 3.3), the event loop of an actor should not be suspended. Instead, one may register a block closure with the future which encapsulates the code to be postponed until the future is resolved. This is done using the `when:becomes:` construct which was first introduced in the E programming language [19].

In the remainder of this section we describe how to integrate future-type message passing in AmbientTalk using the behavioral intercession techniques described in section 4.2.

5.2 Futures

Futures are proxy objects whose message reception semantics deviate from those of normal objects. Rather than implementing such proxies by means of hooks such as Smalltalk's `doesNotUnderstand:` protocol, we implement futures as mirages in order to redefine their default message reception semantics. We describe two changes to the semantics. First, the future's implicit mirror should disallow synchronous method invocations. Second, any asynchronously received message is either buffered if the future is unresolved or forwarded if it is resolved. The code excerpt below shows part of the definition of this future mirror. Asynchronous message reception is reified by means of the `receive` operation.

```
def FutureMirror := extend: actor.defaultMirror with: {
  def state := UNRESOLVED;
  def resolvedValue := nil;
  def inbox := [];
  def invoke(invocation) {
    raise: IllegalOperation.new(
      "Cannot synchronously invoke methods on a future");
    };
  };
  def receive(msg) {
    // msg received by a resolved future?
    if: (state == RESOLVED) then: {
      // forward msg to the resolved value
      msg.sendTo(resolvedValue);
    } else: {
      // buffer message in this future's inbox
      inbox := inbox + [msg];
      nil;
    };
  };
  ...
};
```

The future's implicit mirror is either in an unresolved or in a resolved state, as indicated by its `state` field. Initially, the mirror is unresolved. The transition from an unresolved to a resolved state occurs when an asynchronous `resolve` message is sent to the future's implicit mirror. In addition to the `resolve` method, the future mirror also extends the default metaobject protocol with a `subscribe` method which allows registering closures to be applied when the future has been resolved. These additional methods which are not part of AmbientTalk's default MOP are shown below:

```
def FutureMirror := extend: actor.defaultMirror with: {
  ...
  def subscribers := [];
  def resolve(value) {
    if: (state == UNRESOLVED) then: {
      state := RESOLVED;
      resolvedValue := value;
      // forward all buffered messages
      inbox.each: { |msg| msg.sendTo(value) };
      subscribers.each: { |clo| clo<-apply([value]) };
    };
  };
  def subscribe(closure) {
    if: (state == UNRESOLVED) then: {
      subscribers := subscribers + [closure];
    } else: {
       closure<-apply([resolvedValue])
    }
  }
};
```

When a future is resolved, all messages it accumulated while the result was unavailable will be forwarded to the computed value. Similarly, all subscribed closures are asynchronously applied to the resolved value. Note that the `resolve` and `subscribe` methods reside at the meta level. This stratification of base and meta-level methods has the advantage that metalevel messages are not trapped and forwarded by the `receive` method shown before, as this method only traps messages sent to the base-level future object. The following code excerpt shows the auxiliary methods required to construct and use such a base-level future object.

```
def makeFuture() {
  object: { nil } mirroredBy: FutureMirror;
}
def when: future becomes: closure {
  (reflect: future)<-subscribe(closure);
}
```

Because a future's `subscribe` method resides at the meta level, the `when:becomes:` language construct must send the `subscribe` message to the future's implicit mirror, rather than to the base-level future object itself. This illustrates another advantage of stratifying base and meta-level: base-level messages (sent to the future itself) cannot be mistaken for metalevel messages (sent to the future's implicit mirror). For example, in an application involving newsletters, a `subscribe` message sent to a future for a newsletter object cannot be mistaken for the `subscribe` message which is part of the future's metaobject protocol.

At this point, futures have been introduced as a new data type into the interpreter. However, we have yet to define how futures can be integrated into the actor's message sending protocol. This is the topic of the next section.

5.3 Integration in Message Sending Protocol

In the previous section we have described how to create future mirages based on a mirror object that describes their semantics. In this section, we describe the definition of a custom actor mirror which intercepts both message creation (to attach a future object to the message to capture the return value) and message sending (to return the attached future as a result rather than the default `nil` value). Any base-level asynchronous message send is reified in terms of these two operations by the actor mirror.

The code excerpt below shows the installation of a custom actor mirror which overrides the default `createMessage` and `send` operations. The `createMessage` operation is specialized to return future-type messages, asynchronous messages extended with a `future` field and whose `process` method is overridden. The overridden `process` method will be invoked when the asynchronous message is received and is used to resolve the future with the return value of the invoked method. Finally, the actor's asynchronous message sending semantics is modified by overriding `send`. An asynchronous message send returns a message's associated future rather than the default `nil` value.

```
actor.install: (extend: actor with: {
  def createMessage(sel,args,annotations) {
    // first, create a regular message
    def msg := super.createMessage(sel,args,annotations);

    // if msg was annotated with the OneWayMessage
    // annotation, simply return the regular message
    if: (msg.annotatedAs(OneWayMessage)) then: {
      msg;
```

```
    } else: {
      // turn msg into future-type message
      extend: msg with: {
        // attach a new future to the message
        def future := makeFuture();
        // process is invoked upon reception
        def process(receiver) {
          // delegate to actually invoke the method
          def result := super.process(receiver);
          // resolve the attached future
          (reflect: future)<-resolve(result)@OneWayMessage;
          result;
        };
      }
    };
    def send(msg) {
      def result := super.send(msg);
      if: (!msg.annotatedAs(OneWayMessage)) then: {
        msg.future; // return the message's future
      } else: {
        result;
      };
    };
  }
});
```

Asynchronous messages can be annotated with metadata. In the above code, future-type message passing is disabled for messages annotated as a `OneWayMessage`. This annotation is useful if no return value is required for an asynchronous send. More importantly, the `resolve` meta-message sent to the future mirror requires this annotation to avoid an infinite loop. Without this annotation, the resolution of one future would require the creation of another future, whose resolution requires another future, and so on.

This section presented future-type message passing, an exemplar language construct which relies on behavioral intercession at both the object level (to define the future data type) as well as at the actor level (to integrate futures in the message passing protocol). The next section describes how mirages can be implemented in the language with moderate effect on a system's overall performance.

6. IMPLEMENTATION

As noted by Bracha and Ungar, a desirable software engineering property is that when a feature is not used, it should not incur additional performance penalties [5]. When applied to behavioral intercession, this gives rise to the notion of *partial behavioral reflection* [24]: the principle of limiting the scope of behavioral reflection to where and when it is really needed. AmbientTalk supports two forms of partial behavioral reflection, namely *entity selection* and *operation selection*.

6.1 Entity Selection

Entity selection ensures that metalevel operations on entities which do not use behavioral intercession are not reified. At the language level, AmbientTalk already features a distinction between ordinary objects created using `object:` (which use the default MOP) and mirages created using `object:mirroredBy:` (which have a custom MOP). As a consequence, only metalevel operations invoked on mirages are reified.

The object-oriented AmbientTalk interpreter distinguishes between objects and mirages since they are implemented as distinct classes. As a consequence, when the interpreter invokes metalevel operations (which are implemented as methods on the implementation-level object representation), the dynamic dispatch algorithm of the underlying language is used as a fast test to decide whether a metalevel operation on the receiver should be reified or not.

6.2 Operation Selection

Next to performing entity selection, one may further limit the reification of metalevel operations to only those operations that are actually overridden at the meta level. This is called *operation selection* [24]. In AmbientTalk, such a selection is made possible if the implicit mirror of a mirage is an extension of the `defaultMirror`. By analyzing the methods that the implicit mirror overrides from the `defaultMirror`, we can derive which metalevel operations should be reified, and which operations can proceed natively.

In the implementation of AmbientTalk, operation selection is realized by synthesizing appropriate object representations at runtime. Depending on which metalevel operations need reification, the native methods that implement those operations are replaced by methods which forward a reified operation to the implicit mirror of a mirage.

The current implementation of operation selection in AmbientTalk has some limitations. Currently, the code that analyzes which metalevel operations require a reification assumes that the set of methods overridden by the implicit mirror remains constant. Hence, if the mirror uses e.g. dynamic inheritance to change the set of methods it overrides, additional metalevel operations will not be reified, yielding unexpected behavior.

6.3 Micro-benchmark

The combination of both forms of partial behavioral reflection is depicted in figure 5. Each figure (circle, square, etc.) denotes a particular metalevel operation. A grey background represents no reification (the native implementation), a white background represent reification (forwarding to the mirror). Because of entity selection, native objects can be given a dedicated object representation that does not need to store a reference to a mirror. Mirages, on the other hand, have multiple representations, depending on what metalevel operations need reification. All mirages store a reference to their implicit mirror.

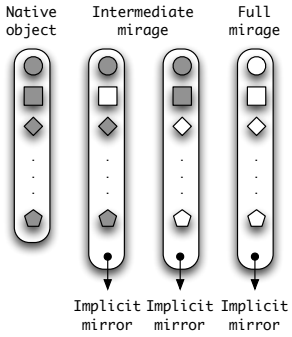

Figure 5: Optimizing the implementation of mirages

We have assessed the performance optimization of partial behavioral reflection in the current implementation by means of a small benchmark. Figure 4 gives an overview of the obtained results. The results show the average running time (in microseconds) to execute the method invocation `obj.m()` on different kinds of objects[2]. The columns distinguish between what kind of partial behavioral reflection is applied. We distinguish three cases: **1)** `obj` is a native object, **2)** `obj` is a mirage mirrored by a mirror that does not

[2]The results shown are obtained by taking the average running time of 10.000 invocations on a Macbook Pro 2.33Ghz Intel Core2 Duo.

	Entity Selection		Entity + Operation Selection	
Native Object	375.3μs	100,0 %	371.5μs	100,0 %
Mirage (default `invoke`)	2207μs	588,06 %	371.4μs	99.97 %
Mirage (custom `invoke`)	2553μs	680,26 %	2524.8μs	679.62 %

Figure 4: Microbenchmark testing the impact of partial behavioral reflection on method invocation.

override `invoke`, **3)** `obj` is a mirage mirrored by a mirror that overrides `invoke` with a dummy method that simply delegates to the native behavior via a super-send.

Although the measured results are obvious, they illustrate that partial behavioral reflection is critical for keeping the performance penalties of behavioral intercession in check. It should be mentioned that our approach to partial behavioral reflection is not the only one to avoid unnecessary reifications. Other techniques such as static analysis or just-in-time compilation can achieve the same goal.

7. DISCUSSION

In this section we briefly discuss how the present architecture differs from the previous version of AmbientTalk [9], as well as related work in the area of behavioral reflection. We end by outlining the current state of AmbientTalk and future work.

7.1 Previous Work

In previous work, we have discussed the metaobject protocol of AmbientTalk/1 – the predecessor of the AmbientTalk language described in this paper – to develop language constructs specifically for mobile ad hoc networks [9]. In AmbientTalk/1, an actor is represented as an active object which executes in a thread of its own, has a message queue and a dedicated *behavior* describing the methods that may be asynchronously invoked on the active object. This behavior object contains base-level application methods *as well as* metalevel methods used to hook into the metaobject protocol. Intercession is made possible by making the active object implement a metalevel method, which is only distinguishable from a base-level method by name.

In AmbientTalk/1 reflection is neither stratified nor encapsulated: base-level code can be affected by the implementation details of metalevel constructs. For example, because the base- and metalevels are not partitioned into separate namespaces, name clashes between the two levels could occur. For example, a base-level method may accidentally be regarded as a metalevel method simply because its name accidentally matches that of a metalevel operation.

7.2 Related Work

Behavioral intercession –that is, the ability of a program to modify its own execution semantics– has been present since the very first work on reflection [23] and its incarnation to object-oriented programming languages [17]. Since then, numerous proposals have been made to introduce behavioral intercession in languages that originally had few (if any) such capabilities.

It is indeed quite rare to see a programming language with a clean reflective architecture for supporting behavioral intercession –such as interception of message sending, object creation, etc.– from the start. A notable exception is the CLOS MOP [13, 21], which can still be considered as the most advanced metaobject protocol in use to date. The difference between the metaobject protocols of CLOS and AmbientTalk is that AmbientTalk's MOP is object-based rather than class-based and that the CLOS metaobject protocol is not entirely stratified [5].

Because the interception of messages sent to objects is a common use case of behavioral intercession, many languages have introduced ad hoc approaches to achieve intercession for this specific case. In Smalltalk, for example, several alternatives have been proposed to control message passing semantics [11], such as method wrappers [6] or using the `doesNotUnderstand:` protocol. In Java, since there is no such thing as a `doesNotUnderstand:` protocol, nor enough reflective facilities to intervene in the method lookup process to define method wrappers, many proposals to introduce behavioral reflection rely on proxies (such as the dynamic proxies added to Java 1.3).

The downside of these approaches is that they implement new metalevel behavior at the base level, thereby violating stratification. For example, when a future is represented as an object overriding `doesNotUnder` or as a dynamic proxy, the future acts as both a base and a metalevel object. Because both levels are indistinguishable, name clashes can occur making it difficult to distinguish between e.g. sending `subscribe` to a future and sending `subscribe` to the object denoted by the future. As exemplified in section 5.2, AmbientTalk's stratified mirror-based MOP avoids such name clashes.

Bytecode transformation is another technique for intervening in the method lookup process of a language [8, 25, 27]. Recently, techniques relying on bytecode transformation have been used to add fine-grained behavioral reflection to Smalltalk [10, 22]. On the one hand, these transformation-based approaches mostly ignore the principles of mirror-based architectures, in particular the issue of structural correspondence: applying standard introspection on transformed code unfortunately reveals the implementation tricks used by the transformation engine. On the other hand, the mirror-based architectures that have been proposed up to now offer only limited behavioral intercession [5]. The architecture presented in this paper precisely reconciles mirrors with behavioral intercession.

Our work also relates to partial behavioral reflection [24]: the principle of limiting the cost of behavioral reflection to where and when it is really needed. We have discussed the implementation of AmbientTalk mirrors and mirages, which support both *entity selection* and *operation selection* [24]. However, AmbientTalk does not support *intra-operation selection*, which is the ability to limit reification to specific *occurrences* of a given operation. This feature is particularly useful for supporting efficiently aspect-oriented extensions [12, 24], and can be provided by the language processor [3].

7.3 Current Status and Future Work

An interpreter for the AmbientTalk language has been implemented in Java[3]. The implementation can run on the Java 2 micro edition (J2ME) platform, under the connected device configuration (CDC).

[3]This implementation can be downloaded at `http://prog.vub.ac.be/amop/at/download`.

Hence, AmbientTalk can be executed on PDAs and high-end cellular phones. Our current experimental setup consists of a number of smartphones which communicate by means of a wireless ad hoc WiFi network.

Currently, AmbientTalk's metaobject protocol reifies among others object instantiation and cloning, object serialization, field and method access, message reception and method invocation. The reflectively implemented futures language construct of which a simpler variant has been discussed in this paper is used as the actual support for future-type message passing in AmbientTalk. The optimizations discussed in section 6 have been achieved in the current Java implementation by generating dedicated Java classes used to represent mirage objects at runtime, using the BCEL bytecode generation toolkit.

Future work focuses on two different uses of the mirror-based architecture. First, we want to employ the architecture to implement more language constructs in the context of mobile ad hoc networks. Second, we would like to apply the mirror-based architecture to develop tool support for AmbientTalk, in the form of e.g. (remote) object inspectors.

8. CONCLUSIONS

AmbientTalk has a mirror-based reflective architecture that supports behavioral intercession. Because of this, AmbientTalk brings the benefits of mirror-based reflection to the realm of reflectively implemented language extensions. First, to meta-level programs, mirrors remain only accessible via the mirror factory, allowing an object to encapsulate its meta-level behavior. Second, implicit mirrors are stratified with respect to base-level code, such that extensions to the metaobject protocol do not interfere with application code. We have illustrated these benefits in a reflective implementation of future-type message passing in AmbientTalk.

AmbientTalk reconciles traditional, structural mirrors with behavioral intercession by dinstinguishing between *explicit* and *implicit* mirrors. Explicit mirrors are used by metaprograms and can only be acquired by means of a mirror factory, which is customizable by the metalevel programmer. Implicit mirrors are used by the interpreter itself in order to intercess metalevel operations on base-level objects. In order to absorb such mirrors, AmbientTalk introduces *mirages*: objects whose MOP is implemented by a causally connected implicit mirror. Finally, AmbientTalk provides support for partial behavioral reflection to minimize the performance penalty for objects which require limited or no support for behavioral intercession.

Acknowledgments.
The authors would like to thank David Ungar for his helpful comments and suggestions for improvement.

9. REFERENCES

[1] AGESEN, O., BAK, L., CHAMBERS, C., CHANG, B.-W., HÖLSZLE, U., MALONEY, J., SMITH, R., UNGAR, D., AND WOLCZKO, M. The SELF 4.1 programmer's reference manual. Tech. rep., Sun Microsystems, Inc. and Stanford University, 2000.

[2] AGHA, G. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] BOCKISH, C., HAUPT, M., MEZINI, M., AND OSTERMANN, K. Virtual machine support for dynamic join points. In Lieberherr [14], pp. 83–92.

[4] BRACHA, G., AND GRISWOLD, D. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications* (1993), pp. 215–230.

[5] BRACHA, G., AND UNGAR, D. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications* (2004), pp. 331–343.

[6] BRANT, J., FOOTE, B., JOHNSON, R., AND ROBERTS, D. Wrappers to the rescue. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP 98)* (Brussels, Belgium, July 1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 396–417.

[7] CAROMEL, D. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming 2*, 4 (November–December 1989), 12–18.

[8] CHIBA, S., AND NISHIZAWA, M. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2003)* (Erfurt, Germany, Sept. 2003), F. Pfenning and Y. Smaragdakis, Eds., vol. 2830 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 364–376.

[9] DEDECKER, J., VAN CUTSEM, T., MOSTINCKX, S., D'HONDT, T., AND DE MEUTER, W. Ambient-oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)* (2006), D. Thomas, Ed., vol. 4067 of *Lecture Notes in Computer Science*, Springer, pp. 230–254.

[10] DENKER, M., DUCASSE, S., AND TANTER, É. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures 32*, 2-3 (July 2006), 125–139.

[11] DUCASSE, S. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming 12*, 6 (1999), 39–44.

[12] HILSDALE, E., AND HUGUNIN, J. Advice weaving in AspectJ. In Lieberherr [14], pp. 26–35.

[13] KICZALES, G., RIVIERES, J. D., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

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

[15] LIEBERMAN, H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications* (1986), ACM Press, pp. 214–223.

[16] LISKOV, B., AND SHRIRA, L. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (1988), ACM Press, pp. 260–267.

[17] MAES, P. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages and Applications* (New York, NY, USA, 1987), ACM Press, pp. 147–155.

[18] MAES, P., AND NARDI, D., Eds. *Meta-Level Architectures and Reflection*. North-Holland, Alghero, Sardinia, Oct. 1988.

[19] MILLER, M., TRIBBLE, E. D., AND SHAPIRO, J. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing* (April 2005), R. D. Nicola and D. Sangiorgi, Eds., vol. 3705 of *LNCS*, Springer, pp. 195–229.

[20] MULET, P., MALENFANT, J., AND COINTE, P. Towards a methodology for explicit composition of metaobjects. In *Proceedings of the 10th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)* (Austin, Texas, USA, Oct. 1995), ACM Press, pp. 316–330. ACM SIGPLAN Notices, 30(10).

[21] PAEPCKE, A. User-level language crafting: Introducing the CLOS metaobject protocol. In *Object-oriented programming: the CLOS perspective*. MIT Press, Cambridge, MA, USA, 1993, pp. 65–99.

[22] RÖTHLISBERGER, D., DENKER, M., AND TANTER, É. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures* (2007). To appear.

[23] SMITH, B. C. Reflection and semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)* (New York, NY, USA, Jan. 1984), ACM Press, pp. 23–35.

[24] TANTER, É., NOYÉ, J., CAROMEL, D., AND COINTE, P. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)* (Anaheim, CA, USA, Oct. 2003), R. Crocker and G. L. Steele, Jr., Eds., ACM Press, pp. 27–46. ACM SIGPLAN Notices, 38(11).

[25] TANTER, É., SÉGURA-DEVILLECHAISE, M., NOYÉ, J., AND PIQUER, J. Altering Java semantics via bytecode manipulation. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)* (Pittsburgh, PA, USA, Oct. 2002), D. Batory, C. Consel, and W. Taha, Eds., vol. 2487 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 283–298.

[26] UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HÖLZLE, U. Organizing programs without classes. *Lisp Symb. Comput. 4*, 3 (1991), 223–242.

[27] WELCH, I., AND STROUD, R. J. Kava - using bytecode rewriting to add behavioral reflection to Java. In *Proceedings of USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)* (San Antonio, Texas, USA, Jan. 2001), pp. 119–130.

[28] YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications* (1986), ACM Press, pp. 258–268.

[29] ZIMMERMANN, C. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.