SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2008; **0**:01–37 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Mirror-based Reflection in AmbientTalk†

Stijn Mostinckx[1]*, Tom Van Cutsem[1], Stijn Timbermont[1], Elisa Gonzalez Boix[1], Éric Tanter[2], and Wolfgang De Meuter[1]

[1] *Programming Technology Laboratory, Vrije Universiteit Brussel, Belgium*
[2] *PLEIAD Laboratory, Computer Science Dept. (DCC), University of Chile, Chile*

SUMMARY

This paper introduces a novel mechanism to perform intercession (a form of reflection) in an object-oriented programming language with the goal of making the language extensible from within itself. The proposed mechanism builds upon a mirror-based architecture, leading to a reusable reflective API that cleanly separates interface from implementation details. However, support for *intercession* has been limited in contemporary mirror-based architectures. This is due to the fact that mirror-based architectures only support reflection explicitly triggered by metaprograms, while intercession requires reflection implicitly triggered by the language interpreter. This work reconciles mirrors with intercession in the context of an actor-based, object-oriented programming language named AmbientTalk. We describe this language's full reflective architecture, highlighting its novel mirror-based approach to reflect upon both objects and concurrently executing actors. Subsequently, we apply AmbientTalk's mirror-based reflection to implement two language features which crucially depend on intercession, to wit future-type message passing and leased object references.

KEY WORDS: Reflection, Metaprogramming, Mirrors, Mirages, Actors, AmbientTalk

1. Introduction

Computational reflection [1, 2] 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 *introspection* if it allows

*Correspondence to: Stijn Mostinckx: smostinc@vub.ac.be
Contract/grant sponsor: S. Mostinckx and S. Timbermont are funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). T. Van Cutsem is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO). E. Gonzalez Boix is funded by the PRFB program of the Institute for the encouragement of Scientific Research and Innovation of Brussels. É. Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and FONDECYT project 11060493.
†This work is based on an earlier work: Mirages: Behavioral Intercession in a Mirror-based Architecture, in Proceedings of the 2007 symposium on Dynamic Languages (DLS 2007) © ACM, 2007. http://doi.acm.org/10.1145/1297081.1297095

programs to inspect the structural aspects of a program. It allows for *self-modification* if programs can modify their structure. It supports *invocation* if base-level operations (e.g. a method call, a field assignment) can be dynamically constructed and executed. It allows for *intercession* if programs can change their behavior, e.g. using custom metaobjects to change the semantics of the language itself [3]. 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, concurrent actor-based object-oriented language. In previous work, we have explicitly presented AmbientTalk as a "language laboratory" for experimenting with novel language features in the context of volatile, ad hoc networks [4]. More concretely, we realized this "language laboratory" by making AmbientTalk a reflective language, such that novel language features can be expressed within the language itself. Whereas our previous metalevel architecture provided adequate support for intercession, it lacked a modular, stratified design. As a result, metalevel extensions to the language could interfere with base-level code and vice versa.

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 reflected upon [5]. Therefore, mostly influenced by Self's mirrors [6], we decided to redesign the AmbientTalk architecture in a mirror-based way. While mirror-based architectures provide proper access to the structure of programs, their support for intercession has been relatively limited. However, intercession is a key enabler for the reflective implementation of language features.

This paper reports on the design of AmbientTalk's mirror-based reflective architecture, which allows reflecting on both objects and actors. In previous work, we have introduced the *mirage*: a base-level object whose semantics are described by a custom *implicit mirror* [7]. The novelty of mirages is that they enable intercession in a mirror-based reflective architecture. The key design issues of a mirror-based architecture and how they are influenced by mirages are recapitulated in this paper (Sections 2.1 and 4.3). In addition, this paper provides a comprehensive overview of AmbientTalk's reflective API on objects (Section 4.1). Also added is a complete description of AmbientTalk's support for reflecting upon actors (Sections 4.2 and 4.4). AmbientTalk's actors are purely event-driven (cf. Section 3.2) and we believe this paper to be the first to discuss a mirror-based "meta-actor protocol" for such actors. We illustrate how AmbientTalk's metaobject and meta-actor protocols can be used to implement two established programming language features (future-type message passing and leased references) as reflective extensions of the base language (Section 5).

Availability An open-source AmbientTalk interpreter with support for mirror-based reflection on both objects and actors is available at `http://prog.vub.ac.be/amop`. The standard library that ships with the implementation contains the complete reflective code for the futures and leased references language features described later in this paper.

2. Mirror-based Reflection

2.1. Design Principles

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 summarise Bracha and Ungar's arguments supporting a mirror-based reflective architecture.

2.1.1. 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, debuggers, object inspectors) against an abstract API, which fosters a higher degree of reusability 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]. Dynamically typed or structurally typed languages (e.g. Strongtalk [8]) inherently avoid such encapsulation breaches.

2.1.2. Stratification

The principle of stratification states that metalevel entities should be cleanly separated from base-level functionality [5]. This separation ensures among others that e.g. a base-level method is not accidentally regarded as part of the metaobject protocol. A stratified design also loosens coupling between the base and the metalevel which 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. `aClass new`) and for metalevel tasks such as code browsing (e.g. `obj class subclasses`). Because of this, it is 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` [6]. 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.

2.1.3. 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 or even marshalled 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 a low-level language, 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].

2.1.4. Summary

An ideal mirror-based system: **1)** provides a reflective API based on interfaces which preserves the encapsulation of metalevel objects; **2)** 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 or to disable reflection when it is not required; **3)** 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; **4)** 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.

2.2. Problem Statement

While mirror-based reflective architectures have traditionally been very successful in providing introspection and self-modification, they have not yet been applied to perform intercession [5]. While introspection and self-modification can be used to build e.g. object inspectors, intercession is required for changing the behavior of an object, e.g. when building novel language constructs. Thus, the problem we address is how to reconcile the above design principles of mirrors with intercession.

In mirror-based systems, a mirror returned by the mirror factory describes the structure of an object (the reflectee), but is otherwise not causally connected with it. That is to say: when the interpreter manipulates the reflectee, it does not do so by consulting the mirror factory and by using the returned mirror. A key design decision when introducing intercession in a mirror-based architecture, therefore, is how to make the interpreter use a mirror to manipulate an object. Should it consult the mirror factory or not? As we will describe in Section 4, we will not make the interpreter consult the mirror factory. Instead, we introduce intercession by distinguishing two kinds of mirrors: explicit and implicit mirrors. We provide a motivation for this decision in Section 6.1.

Explicit mirrors correspond to the traditional mirrors present in Self and Strongtalk. Their goal is to support structural reflection for which no causal connection with the reflectee is required. For example, one can use an explicit mirror to create an "object inspector" for an object stored in a database or for a remote object. Implicit mirrors are "metaobjects" [2, 9] that are truly causally connected to the computation (i.e. used by the interpreter itself), enabling intercession. Like a traditional metaobject, an implicit mirror is tightly coupled to its reflectee (in terms of the implementation, one can imagine the reflectee having a `meta` slot that refers to its implicit mirror). However, *unlike* a traditional metaobject, an implicit mirror satisfies the mirror-based properties of encapsulation and stratification. It satisfies encapsulation because it cannot be accessed from the reflectee directly. Metaprograms must still use the mirror factory. The factory returns the reflectee's implicit mirror by default, but a custom mirror factory can override this policy. Implicit mirrors respect stratification because they remain completely invisible to base-level code.

In Section 4, we discuss implicit and explicit mirrors in detail in the context of the AmbientTalk language. We briefly introduce the base language in the following Section.

3. The AmbientTalk Language

AmbientTalk is a distributed object-oriented programming language. More precisely, the language described here is named AmbientTalk/2 [10], an updated version of the language whose reflective API differs from the version presented in previous work [4]. In the remainder of this paper, we will simply refer to the updated language as AmbientTalk.

Listing 1. A prototypical planar point object

```
def Point := object: {
  def x := 0; // defines a slot named x containing 0
  def y := 0;
  // 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() };
}
def p := Point.new(1,2); // instantiate a new point
```

3.1. Object-oriented Programming in AmbientTalk

AmbientTalk inherits most of its standard language features from Self, Scheme and Smalltalk. From Scheme, it inherits the notion of true lexically scoped closures. From Self and Smalltalk, it inherits an expressive block closure syntax, the representation of closures as objects and the use of block closures for the definition of control structures.

Objects AmbientTalk's objects are reminiscent to those of the prototype-based language Self [11]: classless objects consisting of slots that may contain either regular values or methods. Listing 1 defines a prototypical planar point object. The 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, as shown on the last line. In response to the message **`new`**, an object creates a clone of itself and initializes it by invoking the clone's `init` method. This protocol closely corresponds to that of class instantiation in class-based languages, but rather than allocating a new empty object from a class, a clone is created from a prototype.

When an object receives a message it does not understand, it *delegates* the message to the object bound to its slot named **`super`**. A delegated message is forwarded to another object, but in the subsequent method invocation the **`self`** pseudo-variable will remain bound to the object that originally received the message. Hence, delegation is an object-based alternative to class-based inheritance [12]. A declarative syntax is provided for specifying that a new object delegates to an existing prototype. In the code excerpt below, `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`** slot refers to the parent object and that when a child is cloned, the parent object is cloned as well. Hence, when a `SpatialPoint` is cloned, the clone has its own `Point` parent object with its own copies of the `x` and `y` slots.

```
def SpatialPoint := extend: Point with: {
  def z := 0;
  ...
}
```

Closures 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 used to represent *delayed* computations, such as the branches of an **`if:then:else:`** control structure. 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 use of blocks to define a custom control structure which removes all elements from a collection that fail to satisfy a predicate:

```
def from: collection retain: predicate {
  result := clone: collection; // shallow copy
  collection.each: { |element|
    predicate(element).ifFalse: {
      result.remove(element)
    }
  };
  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: key put: val`) 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.

Type Tags Because AmbientTalk is neither statically typed nor class-based, objects cannot be easily classified. Type tags are a lightweight classification mechanism, used to categorize objects explicitly by means of a nominal type[†]. Type tags are declared using the **`deftype`** keyword and can be a subtype of multiple other type tags:

```
deftype PhotoCopier <: Scanner, Printer;
def CopierPrototype := object: {...} taggedAs: [PhotoCopier];
...
is: CopierPrototype taggedAs: Scanner; // true
```

In the above example, the `PhotoCopier` type tag is defined as a subtype of both the `Scanner` and the `Printer` type tags. The `CopierPrototype` object is explicitly tagged with a `PhotoCopier` type tag. An object can only be tagged when it is created, such that its set of type tags remains constant. The primitive **`is:taggedAs:`** is akin to Java's **`instanceof`** operator and can be used to test whether or not an object is tagged with a certain type tag. Type tags can also be used to annotate methods and messages with metadata, as will be described later.

3.2. Concurrent Programming in AmbientTalk

In AmbientTalk, concurrency is not spawned by means of threads but rather by means of actors [13]. AmbientTalk actors are not "active objects", as they are traditionally represented (e.g. in ABCL [14]),

[†]Although type tags are not used for static type checking, they are best compared with empty Java interface types, like the typical "marker" interfaces used to merely tag objects (e.g. `java.io.Serializable` and `java.lang.Cloneable`).

Figure 1. AmbientTalk actors as communicating event loops

but rather as *communicating event loops*, as is done in the E programming language [15]. An actor is an event loop encapsulating regular objects, which can communicate with one another using either synchronous method invocations (expressed as `o.m()`) or asynchronous message passing (expressed as `o<-m()`). Asynchronous messages are enqueued in an actor's queue of incoming messages, called its *mailbox*. An actor perpetually removes the next message from its mailbox and executes the corresponding method on the receiver of the message. Actors process messages from their message queue serially, i.e. one by one. By processing messages serially, race conditions on the mutable state of the encapsulated objects are avoided.

A reference to an object encapsulated within another actor is called a *far reference*. Message passing via a far reference must be asynchronous. Performing a method invocation via a far reference provokes a runtime exception. Asynchronous messages sent via far references are enqueued in the message queue of the actor that encapsulates the receiver object. Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the control flow of the actors' event loop which perpetually takes messages from their message queue and synchronously executes the corresponding methods on the actor's encapsulated objects. An event loop's control flow never "escapes" its actor boundary. When communication with an object encapsulated by another actor is required, a message is sent asynchronously via a far reference to the object. For example, when `A` sends a message to `B`, the message is enqueued in the message queue of `B`'s actor which eventually processes it.

When sending an asynchronous message to an object that is encapsulated within the same actor, the message is added to the actor's own mailbox and the message's parameters are passed *by reference*, exactly as is the case with regular synchronous message sending. When sending a message across a far reference, objects are instead parameter-passed *by far reference*: the parameters of the invoked method are replaced by far references to the original objects. Objects that have declared themselves to be passed by copy form an exception to this rule and are instead marshalled and passed along in the message. Far references passed back into their originating actor resolve into regular, local references. Far references that are themselves passed by far reference to a third actor are passed unmodified, allowing the third actor to directly communicate with the originating actor.

By default, asynchronous message sends in AmbientTalk have no return value (more precisely, they evaluate to `nil`). However, in section 5.1 we will show how the reflective architecture of AmbientTalk can be used to modify this semantics.

3.3. Distributed Programming in AmbientTalk

In AmbientTalk, two objects are said to be *local* when they are owned by the same actor. Objects are considered *remote* when they are owned by different actors, even if those actors are located on the same machine. By design, AmbientTalk abstracts from the physical location of actors and considers actors to be the unit of distribution. As a consequence of this design, all object references that can span across different machines are far references.

By allowing far references to cross machine boundaries, we must specify their semantics in the face of partial failures. AmbientTalk's far references by default mask partial failures. When a failure occurs, a far reference to a disconnected object starts buffering all messages sent to it. If the network connection is restored at a later point in time, the far reference automatically reconnects and forwards all accumulated messages to the remote object in the same order as they were originally sent. Hence, messages sent to far references are never lost, regardless of the underlying connectivity of the network[‡].

Objects can acquire far references to objects by means of parameter-passing, as described previously. Additionally, an actor can explicitly *export* objects which can then be discovered by remote objects. The details pertaining to this peer-to-peer service discovery protocol and its associated programming language features can be found in previous work [10].

Asynchronous messages may be annotated with type tags by means of the syntax `o<-m()@Tag`. This allows programmers to specify metadata that can be exploited by new language features. Example metadata includes delivery guarantees, message priorities, timeouts for failure handling, etc. We will demonstrate the use of such metadata by the new language features introduced in Section 5.

3.4. Summary

We have introduced the AmbientTalk language, in which computation occurs entirely in terms of classless objects sending messages to one another. Objects are encapsulated within actors. Each actor serializes all access to its encapsulated objects by means of its mailbox. Actors may be distributed across machines and communicate strictly by means of asynchronous message passing. Type tags may be used to annotate messages with additional metadata. With the base language now explained, we can turn our attention to AmbientTalk's metalevel architecture, which is described in the following section.

4. Mirror-based Reflection in AmbientTalk

This section presents the mirror-based metalevel architecture of AmbientTalk. The architecture supports mirrors reminiscent of those in Self and Strongtalk [5], which can be used to perform

[‡]AmbientTalk has been designed as a programming language for use in mobile ad hoc networks, in which network failures are assumed to be omnipresent [16]. Hence the design of making far references resilient to failures by default.

Figure 2. Use of explicit and implicit mirrors.

introspection and self-modification. We name such mirrors explicit mirrors. The novelty of AmbientTalk's metalevel architecture are its implicit mirrors, which can be used to additionally perform *intercession* as well.

The distinction between explicit and implicit mirrors corresponds to the distinction between explicit and implicit reflection proposed by Maes and Nardi [17]. In their proposed distinction, metacomputation triggered by metaprograms which explicitly manipulate mirrors is named explicit reflection. For example, an object inspector could invoke `mirror.listSlots()` to decompose an object into a list of slots (methods and fields) that it can display. Metacomputation triggered implicitly *by the language interpreter itself* is named implicit reflection. For example, source code of the form `o.m()` may lead the interpreter to invoke `mirror.invoke("m")` on a mirror on `o`.

Explicit reflection has been previously explored in the mirror-based architectures of Self and Strongtalk to build such metaprograms as source code browsers, object inspectors and debuggers [5]. Explicit reflection enables introspection, invocation and self-modification, simply by having metaprograms *invoke* the appropriate methods on explicit mirrors. To the best of our knowledge, support for implicit reflection has not been integrated in a mirror-based architecture before. Implicit reflection enables intercession by having metaprograms *specialize* the appropriate methods on implicit mirrors. Figure 2 illustrates the different roles of explicit and implicit mirrors. We further discuss the differences between implicit and explicit mirrors in Section 4.5.2.

Orthogonal to the distinction between explicit and implicit mirrors, AmbientTalk introduces separate mirrors to reflect upon individual objects on the one hand and upon the actor in which they are contained on the other hand. The remainder of this section is organized along the dual distinction between objects versus actors and explicit versus implicit reflection. We conclude this section by highlighting how AmbientTalk's metalevel architecture adheres to the software engineering criteria put forward in Section 2.1 (i.e. encapsulation, stratification and ontological correspondence).

4.1. Explicit Reflection on Objects

A metaprogram (e.g. an object inspector, a debugger, ...) can obtain an explicit mirror on an object to be inspected by invoking the **`reflect:`** function. As in Self, **`reflect:`** consults a mirror factory to create a mirror on the appropriate object. In AmbientTalk, this mirror factory is defined at the level of

the event loop actor that encapsulates the object (cf. Section 4.4). The following code excerpt illustrates how a mirror on the `Point` object `p` introduced in Section 3.1 can be acquired:

```
def mirrorOnP := (reflect: p);
```

A mirror represents the object upon which it reflects as a collection of slots. A slot binds a name to a method. Fields are represented to the programmer as a pair of accessor and mutator slots. The accessor is a nullary method that returns the field value upon invocation. The mutator is a unary method that assigns the field to its single argument upon invocation. Mirrors support introspection (inspection of an object's slots), invocation (reflectively reading and invoking an object's slots) and self-modification (reflectively adding or removing slots). This functionality is illustrated by example in listing 2.

Listing 2. Introspection, invocation and self-modification using mirrors

```
// introspection: list all slots of an object
mirrorOnP.listSlots().map: { |slot| slot.name }; // `[x,y,init,distanceToOrigin,...]
// invocation: reflectively access the contents of a slot
mirrorOnP.grabSlot(`x); // accessor for field x
mirrorOnP.grabSlot(`x:=); // mutator for field x
// invocation: reflectively invoke a method
mirrorOnP.invoke(p, createInvocation(`distanceToOrigin, []));
// self-modification: add and remove slots to/from an object
def [accessor, mutator] := createFieldSlot(`z, 0);
mirrorOnP.addSlot(accessor);
mirrorOnP.removeSlot(`z);
```

Identifiers prefixed with a backquote (`) denote symbols. The first argument to `invoke` denotes the object to which **`self`** will be bound during the method invocation. Note that this object is not necessarily the object which the mirror reflects, because of AmbientTalk's support for object-based delegation. `createInvocation` returns an object encapsulating the selector, actual arguments and potentially other metadata of a method invocation. `createFieldSlot` is a primitive function which, given a name and a value, creates a field and returns two slots representing an accessor resp. mutator method for the field.

AmbientTalk achieves loose coupling between base-level objects and metalevel objects (i.e. mirrors) by enforcing that mirrors are created by means of a mirror factory, rather than making the base-level object refer to its mirror directly. As a consequence, it is possible to create different kinds of mirrors on the same object, depending on the context of use. For instance, one could replace the default mirror factory with a factory that produces mirrors that disallow any modification to base-level objects, effectively "sealing" the structure of those objects.

Listing 3 depicts a factory method that creates "sealed object" mirrors. The factory method creates mirrors which inherit most of their functionality from the mirror returned by the call to **`super`**`.createMirror`. In Section 4.2 we will describe to which object **`super`** refers. For now, it suffices to understand that this call delegates to the default implementation, returning a standard mirror on the object created by the AmbientTalk interpreter. The sealed object mirror overrides the `addSlot` and `removeSlot` metalevel operations such that adding or removing slots reflectively raises an exception.

While listing 3 defines a new factory method, we have not yet shown how this factory method replaces the default mirror factory consulted by the **`reflect:`** function. As mentioned previously, this

Listing 3. A custom mirror factory

```
def createMirror(onObject) {
  extend: super.createMirror(onObject) with: {
    def addSlot(slot) { raise: IllegalOperation.new("Cannot add slot") };
    def removeSlot(slotName) { raise: IllegalOperation.new("Cannot remove slot") };
  }
}
```

mirror factory is defined at the level of each event loop actor in the system. We describe how to replace an actor's default mirror factory in Section 4.4.

4.1.1. API of Explicit Mirrors on Objects

Table I provides a comprehensive overview of the API exposed by explicit mirrors on objects. These methods may be invoked by metaprograms to introspect or modify objects. In fact, these methods may also be invoked by the interpreter and are also part of the API of implicit mirrors, described later in Section 4.3.3. We categorize the different methods defined on mirrors according to different *protocols*. Each protocol reifies different aspects of the object to the metaprogrammer. The remainder of this section briefly discusses each protocol.

Structural Access Protocol Reifies the structure of an AmbientTalk object as a collection of slot objects. An object is said to *own* a slot if the slot is bound directly in the object being mirrored and not in one of the objects in its delegation hierarchy.

Method Invocation Protocol Reifies both synchronous and asynchronous method invocation. When an object `a` executes `b<-m()`, first `a`'s mirror is asked to `send` the message `m`. At a later point in time, when `b`'s actor finally processes the message, `b`'s mirror is asked to `receive` the message. By default, an asynchronously received message is transformed into a regular (synchronous) method invocation (i.e. `b.m()`). This invocation is reified by means of `invoke(b,inv)`. The first argument of `invoke` represents the object to which **`self`** will be bound during method invocation. In case of a delegated message, this is the object that originally received the message, rather than the object in which the method was found. The second argument of `invoke` is an invocation object encapsulating the name and arguments of the method to be invoked.

Object Instantiation Protocol Reifies the act of creating new objects from existing objects. `clone` creates a copy of the mirrored object and recursively clones its parent object. Each object in the cloned parent chain is otherwise a shallow copy of the original. The `newInstance` method also clones the mirrored object but additionally initializes the clone by invoking its `init` method.

Type Testing Protocol Reifies the type tags attached to an object. Objects can be tagged with zero or more type tags. The `isTaggedAs` method is akin to Java's `Class.isAssignableFrom(Class)` method.

`listTypeTags` returns an array of all type tags with which the mirrored object (but not its parent object) has been tagged.

Now that we have discussed explicit reflection on objects in full, we turn our attention to explicit reflection on actors.

4.2. Explicit Reflection on Actors

In Section 3.2 we described that AmbientTalk objects are encapsulated within event loop actors. An object can be encapsulated only in one actor. An actor is responsible for operations that transcend the scope of a single object such as buffering and scheduling asynchronously received messages for its local objects in its mailbox, managing exported objects and the marshalling and unmarshalling of messages (and their arguments) across actor boundaries.

Metaprograms can reflect upon the event loop actor as a whole through an explicit actor mirror, a special object denoting the mirror on the actor. This mirror differs from the explicit object mirrors discussed in the previous section in that it does not reflect upon a single base-level object, but rather upon the event loop actor that is executing the (meta)program. The actor mirror allows manipulating the event loop without exposing its implementation, similar to how a `java.lang.Thread` instance allows manipulating a Java thread without exposing its implementation.

Invoking the top-level function named **`reflectOnActor`** returns an explicit actor mirror on the executing actor. To ensure a loose coupling between the code requiring the actor mirror and its implementation, similar to **`reflect:`** for regular objects, the **`reflectOnActor`** function consults a mirror factory to create the explicit actor mirror. We discuss this actor mirror factory in more detail in Section 4.4.

To exemplify introspection on an actor by means of its explicit mirror, consider the metaprogram in listing 4 that introspects all messages in the actor's mailbox and removes from it all messages matching a given selector, returning them in an array. As a result, these messages will no longer be processed by the actor. This may be useful to implement custom synchronisation policies, e.g. depending on the actor's internal state, it may not be able to process certain messages for a period of time. A copy of the actor's mailbox is acquired by invoking `listIncomingLetters` on the actor mirror. The copy is represented as an array of *letter* objects. Letter objects encapsulate an asynchronous message and the receiver object to which the message is "addressed". Furthermore, letter objects provide a method named `cancel` which can be invoked to remove the letter from the real mailbox before it has been processed.

4.2.1. API of Explicit Mirrors on Actors

Table II provides a comprehensive overview of the API exposed by explicit mirrors on actors. As in the case of explicit object mirrors, these methods may be invoked by metaprograms to introspect on or modify the actor. They are also invoked by the interpreter and are thus also part of the API of implicit actor mirrors, discussed in Section 4.4.1. We briefly discuss each protocol in turn.

Message Sending Protocol Reifies the act of sending an asynchronous message. As seen in the previous section, asynchronous message sending is also reified at the level of the object that sent the

Listing 4. Introspecting on and modifying an actor's mailbox

```
def retractMessagesMatching: selector {
  // introspect on the actor to get a copy of its mailbox
  def mailbox := reflectOnActor().listIncomingLetters();
  // retain only those letters whose message name equals the given selector
  mailbox := from: mailbox retain: { |letter| letter.message.selector == selector };
  // remove all matching letters from the real mailbox
  mailbox.each: { |letter| letter.cancel() };
  // return the removed letters
  mailbox;
}
```

message. By default, the object's mirror delegates this responsibility to the actor's mirror. The default implementation of `send` enqueues the message in the mailbox of the recipient actor.

Message Reception Protocol These methods are the metaprogrammer's interface to the actor's mailbox, which is a queue in which incoming receiver-message pairs (i.e. letters) are buffered before being processed. Letters can be added to the mailbox via the `schedule` method. The `listIncomingLetters` method returns a copy of the mailbox as an array of letter objects. This array is not causally connected to the real mailbox. For example, to remove a letter from the mailbox, one does not remove the letter from the array but rather invokes the letter's `cancel` method. To avoid race conditions, the interpreter guarantees that no messages sent by other actors are added to the actor's mailbox while a (meta)program is executing. The interpreter updates the actor's mailbox only when that actor is not busy processing a message.

Service Discovery Protocol Reifies the act of publishing local or discovering remote objects. The introspective `listPublications` and `listSubscriptions` methods return an array of respectively published local objects and service discovery subscriptions. As in the case of `listIncomingLetters`, these arrays are not causally connected to the implementation-level lists. New publications or subscriptions can be added to the lists by means of the `publish` and `subscribe` methods. Publications or subscriptions may be removed by invoking their `cancel` method, reminiscent of removing a letter from the actor's mailbox. The service discovery protocol forms the basis on top of which more high-level service discovery language features are built [10].

We have now discussed the entire reflective API exposed by explicit mirrors on both objects and actors. In the following two sections, we discuss implicit mirrors on objects and actors.

4.3. Implicit Reflection on Objects

The novelty of AmbientTalk is its support for implicit reflection (i.e. metacomputation triggered by the interpreter itself) in a mirror-based architecture. To support implicit reflection on individual objects, AmbientTalk introduces the concept of a *mirage* object. A mirage is an object whose semantics (in

terms of the methods defined on mirrors) is entirely described by its associated implicit mirror[§]. We first exemplify implicit mirrors and subsequently show how they can become causally connected to mirage objects.

4.3.1. Implicit Mirrors

Consider the prototypical example of logging method invocations performed on an object. It is easy to define a mirror object that overrides `invoke` and performs the logging behavior:

```
def createLogMirror(base) {
  extend: defaultMirror.new(base) with: {
    // override invoke to log the message
    def invoke(delegate, invocation) {
      log("invoked "+invocation.selector+" on "+base);
      super.invoke(delegate, invocation); // default behavior
    };
  }
}
```

The `base` variable refers to the base-level object to be mirrored by the created mirror. Any object can serve as an implicit mirror for a mirage as long as it implements the AmbientTalk metaobject protocol. To facilitate the development of mirror objects that require only small changes with respect to the default language semantics, there exists a top-level variable named `defaultMirror` which is a prototypical implicit mirror object encapsulating AmbientTalk's default metaobject protocol. Most implicit mirrors extend the default mirror to implement their custom semantics.

Mirror objects returned by the above function can perfectly function as explicit mirrors as well. However, if the above mirrors are returned by the actor's mirror factory, only invocations performed *explicitly* upon the mirror are logged (e.g. by evaluating `(reflect: o).invoke(o,invocation)`). When the interpreter is evaluating a standard base-level invocation on the mirror's base object `o` (e.g. by evaluating `o.m()`), no logging happens. This is because the interpreter does not itself consult the mirror factory. Rather, it uses an implicit implementation of the `invoke` operation.

The implicit implementation used by the interpreter can be overridden by making the mirror implicit. In order for the above mirror to become an implicit mirror, it must become *causally connected* to its base-level object. This is the topic of the following section.

4.3.2. Mirages

Mirages are objects which are created by means of the **`object:mirroredBy:`** primitive. Listing 5 redefines the `Point` prototype from Section 3 as a mirage, whose behavior is defined by the implicit mirror defined in the previous section. The **`object:mirroredBy:`** primitive expects two closures as arguments: an *object construction* closure and a *mirror construction* closure. Upon invocation, the primitive creates a new, empty and uninitialized mirage object. It subsequently passes this object to the

[§]The term "mirage" is a pun on the word "mirror" and stresses the idea that, at the implementation level, a mirage is a "ghost" object because its structure and behavior is unknown to the interpreter, as these are specified in the interpreted language itself.

Listing 5. Definition of a mirage

```
def Point := object: {
  def x := 0;
  def y := 0;
  def init(newx, newy) { ... };
  ...
} mirroredBy: { |newMirage| createLogMirror(newMirage) }
```

mirror construction closure. This closure takes the uninitialized mirage as its argument (`newMirage`) and should return its implicit mirror. When a mirror has been created, the **`object:mirroredBy:`** primitive associates the empty mirage with the mirror. From this point on, the mirage and its mirror are causally connected and the implicit mirror is effectively used by the interpreter.

Only after the mirage and the mirror are causally connected is the object construction closure of the object declaration executed. As a result, the construction code is properly reflected by the new implicit mirror. Hence, in listing 5, the field definitions for `x` and `y` are reified as `addSlot` invocations on the logging mirror.

Because they are regular objects, mirages 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.

4.3.3. API of Implicit Mirrors on Objects

The API exposed by implicit mirrors on objects consists of all the methods previously defined on explicit mirrors (cf. table I) and additionally all of the methods defined in table III. Recall that, while introspection, self-modification and invocation are performed simply by invoking these methods, intercession requires the metaprogrammer to implement or override these methods, replacing them with a custom implementation.

With respect to the previously described API in Section 4.1.1, it is worth noting that evaluating base-level code of the form **`def`** `m() {...}` within an object `o` triggers the `addSlot` method of `o`'s implicit mirror, reifying the slot addition. Likewise, base-level code of the form `o.`**`new`**`()` triggers the `newInstance` method on `o`'s implicit mirror. This will clone `o` and subsequently invoke its `init` method with the arguments passed to **`new`**.

Method Invocation Protocol By overriding `invoke`, a metaprogrammer can completely redefine the method invocation semantics of a mirage object. In many cases, the metaprogrammer wants to intervene only in failed method invocations. As in Smalltalk, the `doesNotUnderstand` method is invoked by the AmbientTalk interpreter whenever method lookup has failed, and can thus be used to conveniently ignore, repair or redirect failed method invocations. The default implementation of `doesNotUnderstand` raises an exception.

Figure 3. Accessing explicit versus accessing implicit mirrors.

Object Marshalling Protocol Reifies the act of marshalling and unmarshalling objects when they are passed as the argument of a message sent to another actor. These operations are reified by means of the `pass` and `resolve` methods. Their semantics is akin to Java's `writeReplace` and `readResolve` methods which allow hooking into the Java object marshalling process. The difference between Java and AmbientTalk is that in AmbientTalk, these essentially metalevel operations are properly stratified into a separate mirror object. For pass-by-reference objects (i.e. those who did not declare themselves to be pass-by-copy), the default implementation of `pass` asks the actor to create and return a far reference designating the mirrored object.

4.3.4. Summary

Implicit reflection in AmbientTalk is achieved by means of mirage objects, which are objects associated with a so-called *implicit* mirror. The implicit mirror is used by the interpreter itself when manipulating the mirage. A mirage is constructed in three steps:

1. An empty mirage object is created by the interpreter.
2. An implicit mirror object is constructed and associated with the empty mirage.
3. The empty mirage is associated with its implicit mirror. Subsequently the initialisation code of the mirage is evaluated.

The relationship between mirages, implicit and explicit mirrors is illustrated in figure 3. The figure shows a mirage causally connected to its implicit mirror. Note that the interpreter effectively manipulates the mirage via its implicit mirror, without consulting a mirror factory. Metaprograms, on the other hand, need to pass via a 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 in Section 4.1.

Now that we have discussed implicit reflection on objects in full, we turn our attention to implicit reflection at the actor level.

4.4. Implicit Reflection on Actors

Implicit mirrors on actors allow the metaprogrammer to provide custom implementations for the metalevel operations discussed in Section 4.2. An implicit actor mirror `am` is installed by invoking `reflectOnActor().becomeMirroredBy: am`. The actor mirror plays (among others) the role of mirror factory: it defines a method `createMirror(onObject)` which serves as a factory method for the creation of explicit mirrors on objects. By installing a new implicit mirror on an actor, the metaprogrammer can override this method and thus customize the mirror factory. The code excerpt below illustrates how the sealed mirror factory method defined in Section 4.1 can replace the default factory of the actor.

```
def actorMirror := reflectOnActor();
actorMirror.becomeMirroredBy: (extend: actorMirror with: {
  def createMirror(onObject) {
    extend: super.createMirror(onObject) with: {
      def addSlot(slot) { raise: IllegalOperation.new("Cannot add slot") };
      def removeSlot(slotName) { raise: IllegalOperation.new("Cannot remove slot") };
    }
  }
})
```

Note that the replacement implicit actor mirror extends the current explicit actor mirror such that it inherits the implementation for all other metalevel operations. Only `createMirror` is overridden such that it returns a custom sealed object mirror.

In addition to being a mirror factory for explicit mirrors on objects, implicit actor mirrors also serve as a mirror factory for explicit mirrors on the actor itself. The implementation of `reflectOnActor()` creates an explicit mirror on the actor by invoking a method named `getExplicitActorMirror()` on the implicit actor mirror. By overriding this method, an implicit actor mirror may return a custom explicit actor mirror. This enables loose coupling between client code requiring an actor mirror and the implementation of that mirror. In our current implementation, however, `getExplicitActorMirror` simply returns the implicit actor mirror itself. We have not yet found any convincing examples requiring the creation of a different kind of explicit actor mirror. This may indicate that loose coupling does not seem to be as important at the actor level than it is at the object level.

4.4.1. API of Implicit Mirrors on Actors

The API exposed by implicit mirrors on actors consists of all the methods previously defined on explicit actor mirrors (cf. table II) and additionally all of the methods defined in table IV. We discuss the additional methods below.

Message Sending Protocol The `createMessage` method is invoked whenever an asynchronous message is constructed (e.g. as a result of evaluating code of the form `o<-m()`). It can be overridden to add additional metadata to a message object. Metadata can also be added to the message object by the type tags that are used to annotate it. The default implementation of `createMessage` allows these type tags to add metadata to the message as follows: for each type tag `t` with which a message `msg` is annotated, the implementation invokes `t.annotateMessage(msg)`. The return value of this method should be the message object, potentially extended with the appropriate metadata. We illustrate this protocol by means of a concrete example in Section 5.

Message Reception Protocol The `serve` method is invoked by the interpreter whenever an actor should process a message from its mailbox. The default implementation dequeues a letter object from the actor's mailbox and delivers its message to its associated receiver object. By overriding `serve`, a metaprogrammer can redefine the message processing behavior of an actor. We illustrate this by means of an example, below.

Mirror Creation Protocol Reifies the act of creating explicit mirrors on objects. When calling the **`reflect:`** function to create a mirror on an object, the actor's mirror factory method, named `createMirror` is invoked. This method returns an explicit mirror on the object, whose API is described in Section 4.1.1.

Reference Creation Protocol Reifies the act of creating far references to local objects. The method `createReference` is invoked upon the actor mirror whenever a pass-by-reference object is parameter-passed in between actors. By default the actor returns a far reference to the object, which, as described in Section 3.3, is a remote object reference that masks network failures by default.

Now that the entire API of implicit actor mirrors has been described, we show how they can be used to adapt an actor's message reception protocol in order to support prioritized messages. Consider a type tag `Priority` which can be used to annotate a message send as follows: `obj<-msg(args)@Priority(n)` where `n` is a number denoting the message's priority.

Listing 6 shows a metaprogram that installs a new implicit actor mirror to support prioritized message passing. The implicit mirror overrides all methods of the message reception protocol such that the mailbox of messages is organized as a priority queue sorted according to the messages' priority. The code assumes a function `priorityOf` which retrieves the priority of a message, returning a default value if the message was not explicitly annotated with a priority. Upon installing the new actor mirror, all messages previously scheduled in the old (regular) mailbox are removed and later reinserted in the new (prioritized) mailbox. Recall that the interpreter guarantees that no messages sent by other actors are added to the mailbox while the actor is executing code. Hence, the above code is not subject to race conditions caused by messages arriving while the mailbox is being replaced.

4.5. Evaluation

We have now discussed both explicit and implicit reflection on both objects and actors. In this section, we discuss how AmbientTalk's metalevel architecture upholds the principles of a mirror-based architecture. Furthermore, we summarise the differences between explicit and implicit mirrors.

4.5.1. Mirror-based Reflection

The meta-level architecture of AmbientTalk adheres to the mirror-based reflection principles proposed by Bracha and Ungar [5]. In this section, we illustrate how the reflective API preserves encapsulation, stratification and ontological correspondence.

Encapsulation In Sections 4.1 and 4.2 we have described the reflective API of explicit mirrors on objects and actors, respectively. These APIs are a concrete interface for clients of mirrors. Custom mirrors can be defined and used by metaprograms, as long as they implement AmbientTalk's reflective

API. Because AmbientTalk is a dynamically typed language, any mirror object can be replaced by any other mirror object, as long as both implement the same interface.

Stratification AmbientTalk mirrors are stratified: object mirrors are not accessed directly from the base object on which they reflect. An explicit mirror on an object can only be obtained via the mirror factory. This factory can in turn be replaced by metaprograms. Note that even mirages, which have an associated implicit mirror, are reflected upon by means of explicit mirrors via the mirror factory. For example, a mirror factory may return the sealed object mirror defined in Section 4.1 on the `Point` mirage defined in Section 4.3.2. Hence, mirages enjoy the same loose coupling with their explicit mirrors as any other regular object.

Even though the implicit mirror object must be explicitly tied to the base-level mirage object, base- and metalevel code remain strictly separated (stratified) in different objects. One advantage of this strict separation is that base-level methods cannot accidentally override metaobject protocol methods and vice versa. We illustrate this advantage by means of concrete methods in Section 5.

Finally, stratification is upheld for actor mirrors as well. Explicit actor mirrors are accessed by means the method `getExplicitActorMirror` defined on the implicit actor mirror, which serves as a factory for explicit actor mirrors.

Listing 6. Example of implicit reflection on actors

```
def actorMirror := reflectOnActor();
// cancel all messages currently in the mailbox
def oldMailbox := actorMirror.listIncomingLetters();
oldMailbox.each: { |letter| letter.cancel() };
// install the new actor mirror
actorMirror.becomeMirroredBy: (extend: actorMirror with: {
  def mailbox := PriorityQueue.new();
  def schedule(rcv, msg) {
    def letter := object: {
      def receiver := rcv;
      def message  := msg;
      def cancel() { mailbox.remove(self); }
    };
    mailbox.enqueue(priorityOf(msg), letter);
  };
  def serve() {
    def letter := mailbox.dequeue();
    if: (letter != nil) then: {
      letter.message.process(letter.receiver)
    };
  };
  def listIncomingLetters() { mailbox.toArray() };
});
// reschedule all messages in the new mailbox
oldMailbox.each: { |letter|
  actorMirror.schedule(letter.receiver, letter.message)
};
```

Ontological Correspondence AmbientTalk's mirror architecture is structurally correspondent to the base-level: all base-level computation is modelled in terms of methods defined on mirrors. Upholding structural correspondence required us to define mirrors on actors as well as on regular objects, because base-level AmbientTalk computation cannot be described in terms of objects alone. Actors are a crucial part of the base-level computation, and thus require a mirrored representation as well, even though actors have no object representation at the base-level.

The issue of requiring a separate API for high-level and low-level language does not apply to our current implementation of AmbientTalk: the interpreter uses the parse trees themselves as instructions to evaluate method bodies, hence there is no low-level language (e.g. bytecode) to reflect upon.

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

4.5.2. Properties of Explicit and Implicit Mirrors

AmbientTalk's metalevel architecture allows metaprogrammers to define their own implicit and explicit mirrors. However, the architecture does assume important substitutability requirements upon objects representing implicit or explicit mirrors which have been left unexplained up to this point:

Substitutability of implicit mirrors: *An implicit mirror is required to be substitutable for the default mirror.*

This requirement implies that implicit mirrors must provide a *complete* implementation of the metaobject protocol. It is motivated by the fact that implicit mirrors are used by the interpreter which can invoke any method of the metaobject protocol. This requirement is usually, but not necessarily, upheld by the metaprogrammer by having a custom implicit mirror delegate to the `defaultMirror`, which already provides a complete implementation of the MOP.

Substitutability of explicit mirrors: *An explicit mirror on an object is required to be substitutable for the implicit mirror of that object.*

This requirement implies that an object's explicit mirror can be used everywhere a metaprogrammer expects its implicit mirror. It is motivated by the fact that metaprograms that want to access an object's implicit mirror must always do so *via* an explicit mirror because the principle of stratification forces them to use the mirror factory. This requirement is usually, but not necessarily, upheld by the metaprogrammer by having an object's explicit mirror delegate to the object's implicit mirror, such that the explicit mirror is guaranteed to provide at least the interface of the object's implicit mirror. The default mirror factory returns the implicit mirror of a mirage as its default explicit mirror. An implicit mirror can thus always be used as an explicit mirror.

A direct corollary of the above requirements is that an explicit mirror must also provide a complete implementation of the MOP. The requirements also state that if an implicit mirror adds methods to an object's MOP, the explicit mirror must also support them. However, methods added to the MOP of an object by an explicit mirror must not necessarily be supported by the object's implicit mirror. In Section 5.1 we will describe a metaprogram that depends upon substitutability of explicit mirrors for its correct operation.

4.6. Summary

AmbientTalk's mirror-based architecture distinguishes explicit from implicit reflection. Explicit reflection, as required by metaprograms such as object inspectors, debuggers, etc. is achieved simply by explicitly invoking the methods of mirrors on objects and actors. Implicit reflection, as required for tracing and monitoring objects or for implementing language features, is achieved by having metaprogrammers define their own *implicit* mirrors on objects and actors. Implicit mirrors often specialise the default implementations provided by the language. An implicit mirror on an object is installed when the object is created (by means of `object:mirroredBy:`) while an implicit mirror on an actor can be installed at any point in time (by means of `becomeMirroredBy:`). Table V summarizes all reflective operations to retrieve and install mirrors.

5. Growing a Language using Intercession

In this section, we illustrate how AmbientTalk's reflective facilities, described in the previous section, can be used to "grow the language" [18] with more expressive language features. The language features discussed in this section augment the basic AmbientTalk language presented in Section 3 as follows:

- We extend the language with *future-type message passing* [14] in which asynchronous message sends can be made to return a future rather than the default `nil` value. A future is a placeholder for the return value that is being computed asynchronously and avoids the use of separate callback methods to process results.
- We extend the language with *leased far references* in which far references are combined with leasing. Leasing [19] is a time-based technique by means of which both client and server objects can deal with long-lasting network partitions.

Both language features require explicit support for intercession (in particular the ability to intercept messages). Both features are implemented using a two-step methodology:

1. We use implicit mirrors on objects to introduce a new data type into the language (a future resp. a leased far reference).
2. We use implicit mirrors on actors to introduce these data types in the relevant actor-level (the message sending resp. reference creation protocol).

We describe the concrete implementation of both language features in each of the following sections.

5.1. Future-type Message Passing

Future-type message passing is an exemplar language feature which relies on 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 introduction of future-type message passing reintroduces the notion of a return value into a system with asynchronous message passing.

In AmbientTalk, an asynchronous message send normally has no return value (i.e. it evaluates to `nil`), forcing the programmer to rely on explicit, separate callback methods to obtain the result

of an asynchronous computation. Future-type message passing is a classic technique to reconcile asynchronous message sends with return values, by making an asynchronous send immediately return a *future* object [14]. 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 := dbms<-connect(properties);
def employees := database<-query("SELECT * FROM Employee");
when: employees<-nextRow() becomes: { |employeeRow|
  system.println(employeeRow.name);
}
```

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. This future object will be *resolved with* an object representing a connection to the database at a later point in time. Subsequently, an asynchronous `query` message is sent to the `database` future, which will buffer the message and forward it to its resolved value once this value is available. Only asynchronous messages can be sent to a future object. Invoking a method or accessing a field synchronously on a future raises a runtime exception.

In the majority of systems introducing futures, when code requires access to the actual resolved value of the future, the thread requiring the value is suspended until the future is resolved. Such a synchronisation style is called *wait-by-necessity* [20]. However, because AmbientTalk actors are event-driven (as explained in Section 3.2), the event loop of an actor should never be suspended in the middle of a computation. Instead, one can register a block closure with the future which represents a "continuation": it encapsulates the code to be postponed until the future is resolved. This is done using the **`when:becomes:`** control structure which was first introduced in the E programming language [15]. The closure passed to **`when:becomes:`** will be applied to the future's resolved value.

In the remainder of this section we describe how to integrate future-type message passing in AmbientTalk by means of implicit mirrors on objects and actors. We focus on the key issues of the language abstraction. The actual implementation of future-type message passing shipped with the AmbientTalk distribution covers additional features such as exception handling and timeouts.

5.1.1. Future Data Type

Futures are proxy objects whose message reception semantics deviate from that of normal objects. Rather than implementing such proxies by means of hooks such as Smalltalk's `doesNotUnderstand:` protocol [21], we implement futures as mirages such that their message reception semantics can be modified by an implicit mirror. We describe two changes to the semantics. First, the future's implicit mirror disallows synchronous method invocations by overriding `invoke`. Second, any asynchronously received message is either buffered if the future is unresolved or forwarded if it is resolved. This is done by overriding `receive`.

Listing 7 shows the definition of a future's implicit mirror. The future is either in an unresolved or in a resolved state, as indicated by the `isResolved` variable. Initially, the future is unresolved. The transition from an unresolved to a resolved state occurs when an asynchronous `resolveWithValue` message is sent to the mirror. In addition to the `resolveWithValue` method, the mirror also extends the default metaobject protocol with a `subscribe` method which allows registering observers to be notified when the future has been resolved. When a future is resolved, all messages it has accumulated while it

Listing 7. Implicit Mirror on Futures

```
def createFutureMirror(base) {
  // variables private to the mirror
  def isResolved    := false;
  def resolvedValue := nil;
  def inbox         := Vector.new();
  def observers     := Vector.new();
  extend: defaultMirror.new(base) with: {
    def invoke(delegate, invocation) {
      raise: IllegalOperation.new(
        "Cannot synchronously invoke methods on a future");
    };
    def receive(msg) {
      // msg received by a resolved future?
      if: (isResolved) then: {
        // forward msg to the resolved value
        reflectOnActor().send(resolvedValue,msg);
      } else: {
        // buffer message in this future's inbox
        inbox.append(msg);
      };
    };
    // methods added to a future's MOP
    def resolveWithValue(value) {
      if: !(isResolved) then: {
        isResolved := true;
        resolvedValue := value;
        // forward all buffered messages
        inbox.each: { |msg| reflectOnActor().send(value,msg) };
        observers.each: { |obs| obs<-notifyResolved(value) };
      };
    };
    def subscribe(observer) {
      if: (isResolved) then: {
        observer<-notifyResolved(resolvedValue);
      } else: {
        observers.append(observer);
      }
    }
  };
};
```

was unresolved are forwarded to the computed value (in the same order as they were received by the future). Similarly, all subscribed observers are asynchronously notified of the resolved value.

We now turn our attention to the so-called "language constructs" which form the language feature's public interface to the base-level programmer. The following code excerpt shows how the programmer may create a base-level future object.

```
def createFuture() {
  def future := object: { } mirroredBy: { |base| createFutureMirror(base) };
```

```
  def resolver := object: {
    def resolve(value) { (reflect: future).resolveWithValue(value) };
  };
  [ future, resolver ] // return a tuple containing the future and the resolver
};
```

A future is represented as an empty mirage object with an implicit future mirror. `createFuture` returns two values: the future itself and an associated *resolver* object. The resolver has a single method named `resolve` which can be used by the base-level programmer to resolve the future without having to know anything about the meta-level interface of the future. Below is the definition of the **`when:becomes:`** control structure that allows base-level programmers to postpone the execution of a closure until a future has been resolved.

```
def when: future becomes: closure {
  (reflect: future).subscribe(object: {
    def notifyResolved(value) { closure(value) }
  });
};
```

Both the `resolveWithValue` and `subscribe` methods defined previously are part of the *mirror* on the future and thus reside completely at the metalevel. This stratification of base and metalevel 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. Because `resolveWithValue` and `subscribe` reside at the meta-level, the language constructs defined above must invoke these methods on the future's *mirror*, rather than on 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.

A subtle point in both of the above code snippets is that **`reflect:`** is used to acquire a mirror on the future. Because **`reflect:`** consults the actor's mirror factory, the explicit mirror being returned may not be the future mirror defined in listing 7. However, because of the substitutability requirement on explicit mirrors (cf. Section 4.5.2), the metaprogrammer of the above language constructs may assume that the explicit mirror adheres to the interface of the future's implicit mirror. Thus, the metaprogrammer may invoke additional methods such as `resolveWithValue` and `subscribe` on the explicit mirror.

Note that because our architecture explicitly distinguishes explicit from implicit mirrors, we introduce the freedom to choose whether reflecting upon a resolved future returns a mirror on the future itself (by default, the object returned by `createFutureMirror`) or a mirror on the value with which the future is resolved (which would make metaprograms oblivious to resolved futures).

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 automatically attached to asynchronous messages. This is the topic of the following section.

5.1.2. 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 how futures can be integrated in the AmbientTalk

Listing 8. Integrating futures in the message sending protocol

```
deftype OneWayMessage <: AsyncMessage;
deftype FutureMessageTag <: AsyncMessage;
def FutureMessage := extend: FutureMessageTag with: {
  def annotateMessage(msg) {
    def [future, resolver] := createFuture();
    extend: msg with: {
      // invoked whenever msg is sent to a receiver object
      def sendTo(receiver, sender) {
        super.sendTo(receiver, sender); // returns nil by default
        future // return the future instead
      };
      // invoked whenever msg arrives at a receiver object
      def process(receiver) {
        def result := super.process(receiver); // invoke the method
        resolver<-resolve(result)@OneWayMessage;
        result;
      };
    };
  };
};
```

message sending protocol. First of all, we describe how futures can be added to individual messages by annotating them with a new type tag. Subsequently, we describe how to automatically add this type tag to *every* message send performed within an actor by means of a custom implicit mirror on the actor.

Listing 8 shows the definition of two new type tags, `OneWayMessage` and `FutureMessage`, which can be used to annotate asynchronous message sends. The `OneWayMessage` annotation can be used to distinguish asynchronous messages which have no (meaningful) return value and therefore never need to resolve a future. When a message is annotated with the `FutureMessage` type tag, this signals that the programmer is interested in the return value of that asynchronous message. Therefore, the message needs to be equipped with a future and the necessary infrastructure to resolve the future when the result has been computed. This is done by having the type tag override the `annotateMessage` method. Recall from section 4.4.1 that this method is invoked when the type tag is used to annotate an asynchronous message. The method returns the original message, extended with the necessary metadata.

The extended message object overrides the original message's `sendTo` and `process` methods. The former method is invoked when the asynchronous message is sent and is specialized such that it returns the attached future rather than the default value (**`nil`**). The `process` method is invoked by the recipient actor when the message is received and is used to resolve the future with the return value of the invoked method. This method is executed remotely, on a parameter-passed copy of the message. The `resolver` object, being an implicit parameter of the message, is passed by far reference. This explains why the `resolve` message is sent asynchronously using `<-`. This message is also explicitly annotated as a `OneWayMessage`. Without such an annotation, if future-type message passing is installed as the default (see below), resolving one future would require the creation of another future. Resolving that future would again create another future, and so on.

The introduction of the `FutureMessage` type tag allows the programmer to explicitly annotate which messages require a future. Installing future-type message passing as the *default* message passing semantics can be achieved by means of a custom actor mirror. The code excerpt below installs a custom actor mirror which intercepts message creation to implicitly add the `FutureMessage` annotation to all newly created messages.

```
def actor := reflectOnActor();
actor.becomeMirroredBy: (extend: actor with: {
  def createMessage(selector, args, annotations) {
    if: !((annotations.contains: { |tag| tag.isSubtypeOf(FutureMessage) }).or:
         {(annotations.contains: { |tag| tag.isSubtypeOf(OneWayMessage) }) }) then: {
      annotations := [FutureMessage] + annotations;
    };
    super.createMessage(sel, args, annotations);
  }
})
```

Whenever a new asynchronous message object is created, the above actor mirror automatically adds the `FutureMessage` annotation unless the message was already annotated as a `FutureMessage` or it was explicitly annotated as a `OneWayMessage`. The check for the `OneWayMessage` annotation ensures that messages tagged as being "one-way" never get associated with a future.

This concludes our implementation of the future-type message passing language featue. In the following section, we repeat the methodology applied in this section to another language feature.

5.2. Leased Far References

We now present *leased far references*, a language feature that limits the lifetime of far references such that the object being referred to by a far reference (henceforth called the *service* object) and the objects using the far reference (henceforth called the *client* objects) can deal with potentially permanent network partitions.

As explained in Section 3.3, AmbientTalk's far references by default mask partial failures: messages may be sent to a disconnected far reference, where they will be buffered until the far reference becomes reconnected. Consequently, transient network partitions have no direct impact on the application. Computation is resumed transparently, allowing both client and service objects to continue their collaboration from the point where they disconnected. This behavior is desirable in mobile ad hoc networks since they exhibit more frequent transient network partitions than traditional computer networks. However, not all network partitions are transient. Some partitions may result in permanent failures, e.g. when a device has crashed or has moved out of wireless communication range and never returns.

To preserve the resilience of far references to transient failures while still being able to deal with permanent failures, AmbientTalk employs *leasing* [19]. A *lease* denotes the right to access a resource (e.g. an object) for a finite amount of time. At the discretion of the owner of the resource a lease can be renewed, prolonging access to the resource. In AmbientTalk, we chose to represent leases as a special kind of far references which we name *leased* far references. A leased far reference behaves like a far reference, except that it grants access to its service object only for a limited period of time. Moreover, whenever a message is sent via a leased far reference, the lease is transparently renewed. By automating the renewal of the lease upon message sending, tedious boilerplate renewal code is avoided.

Listing 9. A leased session in an online shopping application

```
def openSession() {
  def shoppingCart := Cart.new(); // to store purchased items
  def session := object: {
    def addItemToCart(anItem) { ... }
    def checkOutCart() { ... }
  };
  def leasedSession := lease: 5*60*1000 for: session;
  when: leasedSession expired: {
    ... // free up resources used by this session
  };
  leasedSession; // return lease on the session to the client
};
```

Listing 9 illustrates leased far references in the context of an online shopping application. In the example, a client can ask a server to start a shopping session by sending it the `openSession` message. In response to this message, the server returns a session object which implements methods that allow a client to place items in its shopping cart or to check out. It is assumed that the client uses future-type message passing to get a reference to the `openSession` method's return value. If the `session` object would be returned directly, the client would acquire a far reference to it. The method instead returns a *leased* far reference to the session, to ensure that the client's connection to the session is leased.

The **`lease:for:`** function takes as parameters a time interval (in milliseconds) and the service object to which it grants access, and returns a leased far reference that remains valid for the indicated time interval (5 minutes in the example). The **`when:expired:`** function implements a control structure that applies a closure after a given leased reference expires. In the example, this control structure is used to schedule clean-up code when the session expires. At client side, a customer can ask a server to open a shopping session as follows:

```
def mySession := server<-openSession()@FutureMessage;
...
mySession<-addItemToCart(selectedItem);
```

`mySession` stores a leased far reference to a session object which remains valid for the next 5 minutes. After this time period, access to the session is terminated and the leased far reference is said to *expire*. As discussed above, a leased far reference is automatically renewed whenever a message is successfully delivered to its service object. Hence, as long as a client actually uses the session (by adding items to its cart or by checking out), the session remains active. Note that, because we chose to model leasing by means of a special kind of far reference, the client can use the leased reference as if it were the session object itself. The use of leasing is made transparent to the client.

Upon a network partition, the leased far reference cannot successfully deliver messages to the session and as a result it cannot be renewed. If the network partition outlasts the reference's lease period, the reference will expire and the logical connection between client and service is permanently broken. Both client and service objects can schedule clean-up actions with the leased reference upon expiration. Hence, both sides can gracefully deal with the termination of the logical connection. In particular, leasing provides fault-tolerant garbage collection of service objects: once all leased references to a

service object have expired, this object is subject to garbage collection (provided it is not referred to by any local objects).

In the remainder of this section we describe how to implement leased far references reflectively in AmbientTalk. Leased far references require intercession at the object-level (to intercept messages sent via a leased reference). Furthermore, if we want to make leased references the default type of reference used for inter-actor communication, intercession at the actor-level is required (to integrate leased references in the actor's reference creation protocol).

5.2.1. Leased Far Reference Data Type

Like futures, we implement leased far references as mirages whose implicit mirror mimics the behavior of far references with some modifications to introduce the leasing semantics. Listing 10 shows the definition of a leased reference's implicit mirror. `base` is the base-level representation of the leased far reference. It is simply a proxy object with special message passing semantics. `serviceObj` is the service object to which the leased reference grants access. The `isExpired` variable holds the leased reference's state (i.e. expired or not). When the mirror is initialized, a `timer` object is created by means of the **`when:elapsed:`** control structure which takes a time interval (in milliseconds) and a closure as parameters and applies the closure once the time interval has elapsed. When the `timeInterval` elapses, the leased reference expires.

Similar to far references, leased far references can only carry asynchronous messages. To enforce this, the leased reference's implicit mirror disallows synchronous method invocations by overriding the `invoke` MOP method, causing it to raise a runtime exception. The mirror also overrides the `receive` MOP method to intercept all asynchronous messages. As long as it has not expired, the mirror renews the lease and forwards the message to the service object. For all other MOP methods, the leased reference mirror inherits the default implementation from `defaultMirror`.

The leased reference's implicit mirror also extends the default metaobject protocol with methods that manage the life cycle of a leased reference. The `expire` method terminates remote access to the leased reference by invoking a primitive method named `takeOffline`. This primitive removes the leased reference from the export table of the owning actor. This table is used by the interpreter to resolve far references into local object references. When an object is removed from the export table, it is no longer accessible remotely and, importantly, becomes subject to garbage collection once it is no longer locally referenced. The `renew` method renews the lease by resetting the `timer` object that refers to the code scheduled to expire the reference. Similar to the mirror on a future, the leased reference mirror defines an `addExpiredObserver` method which allows registering observers to be notified upon expiration.

Finally, a leased far reference mirror overrides `pass` to modify the parameter-passing semantics of its base-level object. When a leased reference is parameter-passed, it is not passed by far reference but rather by copy: the remote client will receive its own, local leased far reference proxy that allows it to locally keep track of the lease time left. The parameter-passing semantics for leased far references is explained in further detail in the following section.

Listing 11 shows the definition of the language constructs by means of which the programmer can use leased far references. A leased far reference is represented as an empty mirage object with an implicit leased reference mirror. The **`when:expired:`** control structure allows programmers to postpone

Listing 10. Implicit Mirror on Leased References

```
def createLeasedRefMirror(base, serviceObj, timeInterval) {
  def isExpired := false;
  def timer;
  def expiredObservers := Vector.new();
  extend: defaultMirror.new(base) with: {
    timer := when: timeInterval elapsed: { self.expire() };
    def invoke(delegate, invocation) {
      raise: IllegalOperation.new(
        "Cannot invoke a method synchronously on a leased reference");
    };
    def receive(msg) {
      if: !(isExpired) then: {
        self.renew();
        reflectOnActor().send(serviceObj, msg); // forward msg to the service object
      };
    };
    def expire() {
      if: !(isExpired) then: {
        isExpired := true;
        takeOffline(self); // terminate remote access to the leased reference
        expiredObservers.each: { |obs| obs<-notifyExpired() }
      };
    };
    def renew(renewalTime := timeInterval) { // timeInterval is a default argument
      // reset the timer by cancelling the previously scheduled closure
      timer.cancel();
      timer := when: renewalTime elapsed: { self.expire() };
    };
    def addExpiredObserver(observer) {
      if: isExpired then: {
        observer<-notifyExpired();
      } else: {
        expiredObservers.append(observer);
      };
    };
    def pass() { ... }
  };
};
```

the execution of a closure until a leased far reference expires. It is implemented by registering an observer on the leased reference's mirror.

We have shown how leased far references can be reflectively implemented as proxy objects with a custom message passing semantics, by means of implicit mirrors. However, the programmer must still manually wrap service objects using the `lease:for:` language construct. In the following section, we show how leased far references can be introduced automatically by hooking into the actor's reference creation protocol.

Listing 11. Language constructs to manipulate leased far references

```
def lease: period for: serviceObj  {
  object: {} mirroredBy: { |base| createLeasedRefMirror(base, serviceObj, period) };
};
def when: leasedRef expired: closure {
  (reflect: leasedRef).addExpirationObserver(object: {
    def notifyExpired() { closure() }
  });
};
```

5.2.2. *Integration in Reference Creation Protocol*

In the previous section, we have described how to create leased far references based on a mirror object that describes their semantics. We also described how leased far references can be added to individual objects by means of the `lease:for:` construct. In this section, we describe how to automatically add a leased far reference to *every* pass-by-far-reference object which is parameter-passed in between actors (and thus becomes remotely accessible). More concretely, we define an implicit actor mirror which intercepts the act of creating a far reference to a local object to create a leased far reference instead.

Listing 12 shows the definition of the new implicit actor mirror which overrides the default `createReference` method. Recall from Section 4.4 that pass-by-far-reference parameter passing is reified by invoking this method on the actor mirror. The method is specialized to attach a leased far reference to the object, valid for a default time interval (specified by the constant `DEFAULT_LEASETIME`).

Listing 12. Integrating leasing in the reference creation protocol

```
def actor := reflectOnActor();
actor.replaceMirror: (extend: actor with: {
  def createReference(toObject) {
    lease: DEFAULT_LEASETIME for: toObject;
  };
});
```

The object returned from `createReference` must itself be parameter-passed as well. As described in the previous section, leased far references are not pass-by-far-reference and thus do not recursively trigger the `createReference` method. Rather, a leased far reference is passed by copy. Therefore, a leased far reference consists of two leased reference mirrors: one at client-side and one at server-side. A client-side mirror behaves slightly different than its server counterpart (which is the one discussed previously). The key difference is that it does not actually grant access to the service object itself but rather to the server-side mirror. Messages intercepted by the client-side mirror are thus forwarded to the server-side mirror, which then forwards them to the real service object. The client-side mirror also maintains its own timer (which is weakly synchronized with the timer of the server mirror) and its own `when:expired:` observers (which allow client objects to be notified upon the expiration of the leased reference without requiring a network connection with the server).

5.3. Summary

In the above sections, we have shown how to incorporate new language features in the AmbientTalk base language. They illustrate that AmbientTalk is indeed an extensible language. Both language features require support for intercession: futures and leased far references are proxies which require the ability to intercept messages to alter their message passing semantics. Futures postpone the processing of messages until the future is resolved while leased far references renew a lease upon each message send. Support for intercession was provided by means of a two-step methodology:

- Implicit mirrors on objects have been used to implement the proxy objects as mirages. Because futures and leased far references are mirages, the metaprogrammer has total control over the semantics of message delivery via their implicit mirror.
- Implicit mirrors on actors have been used to integrate the language feature with existing language features. In the case of futures, we have shown how the actor's message sending protocol can be adapted to automatically include a future in each asynchronous message send. In the case of leased far references, we have shown how the actor's reference creation protocol can be adapted to automatically parameter-pass objects in between actors "by leased reference" rather than "by far reference".

The major advantages of the mirror-based approach are that the implementation of the language features is *encapsulated* (it does not disrupt other metaprograms, e.g. a future or leased far reference can be inspected by an object inspector as any other regular object) and *stratified* (base-level messages pertaining to the application are not misinterpreted as meta-level messages pertaining to the language constructs and vice versa).

6. Discussion

In this section, we motivate the more important design decisions of AmbientTalk's mirror-based architecture as it is presented in Section 4.

6.1. Distinguishing Explicit from Implicit Mirrors

AmbientTalk enables intercession in a mirror-based architecture by introducing the concept of an implicit mirror, which is associated with a so-called mirage object. An implicit mirror is causally connected to its mirage and is thus very similar to the traditional notion of a metaobject [2, 9]. Unlike a metaobject, an implicit mirror remains encapsulated and stratified. While the interpreter can directly access a mirage's implicit mirror, other metalevel programs must use the mirror factory. The default mirror factory returns an object's implicit mirror, but this policy can be changed by metaprograms.

In our current architecture, explicit mirrors (as defined by metaprograms and returned from custom mirror factories) are not causally connected to the base-level computation. They could be made causally connected if the interpreter is treated like any other metaprogram that must consult the mirror factory when manipulating an object. It would then be possible to make the interpreter use an explicit mirror. We did not consider this alternative architecture because of a number of reasons:

- An explicit mirror does not necessarily reflect upon a concrete base-level object, such that there is no object for the mirror to become causally connected to. For example, an explicit mirror could reflect upon an object stored in a database. The explicit mirror would then interface with the database to represent the relational data as an object (i.e. a collection of slots), which is useful for other metaprograms such as an object inspector, which can then inspect persistent objects. In this case, no causal connection to any concrete base-level object is required.
- Objects can be reflected upon by *multiple* and *unrelated* explicit mirrors, each providing a different form of reflective access to its reflectee. 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. If we were to causally connect the proxy to an explicit mirror, which one should the interpreter use? Such situations are avoided by enforcing a strict one-to-one correspondence between an object and its *unique* implicit mirror[¶].
- Making the interpreter use explicit mirrors may cause unwanted interference. Consider the sealed object mirror introduced in Section 4.1: it can ensure read-only reflection by metaprograms such as object inspectors; however if it is used by the interpreter, the interpreter *itself* would be precluded from adding slots to an object, making it impossible to instantiate non-empty base-level objects. Such issues are avoided by distinguishing *explicit mirrors* for use by metaprograms (e.g. the sealed object mirror) from *implicit mirrors* for use by the interpreter (e.g. the log mirror).

Even though our current architecture does not allow an explicit mirror to become causally connected to its reflectee, we acknowledge that it may useful for an explicit mirror to react upon the manipulation of its reflectee by the interpreter. A typical example would be an object inspector that needs to update its graphical user interface whenever the value of one of its reflectee's slots is changed. Currently, we must implement such an inspector by means of an implicit mirror that can override the necessary MOP methods. In future work, we would like to extend the reflective API with methods that allow explicit mirrors to register observers on their reflectee, allowing them to get notified of the execution of a meta-level operation without actually having to override it.

In general, implicit mirrors are most often useful to encode "language features": they change the structure or behavior of an object as viewed by the interpreter itself, thus changing the semantics of the language. We have provided concrete examples of such mirrors in section 5. If the goal is only to change the structure or behavior of an object for other metaprograms (e.g. for the purposes of displaying an object stored in a database), then explicit mirrors are sufficient.

6.2. When and How to Connect Implicit Mirrors

The reader may have noticed that there is a difference in the way implicit mirrors are causally connected with objects versus with actors. Contrary to implicit mirrors on objects, which are created when the mirage object is declared (by means of `object:mirroredBy:`), implicit mirrors on actors can be installed at any time during the lifetime of the actor (by means of `becomeMirroredBy:`). There is no `actor:mirroredBy:` construct to statically connect an actor with an actor mirror. This design is

[¶]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 [22].

motivated by the fact that the code that creates an actor may be totally independent of the code that the actor will execute. As such, the creator of the actor may be unaware of reflective extensions required by the code to be executed. This is particularly the case in frameworks that spawn generic actors, e.g. a unit testing framework that spawns an actor to run an arbitrary unit test.

Similarly, one may wonder why there is no support for `becomeMirroredBy:` on regular objects. Using such a construct, a regular object could be turned into a mirage and vice versa. We opted to disallow this because it enables the interpreter to in-line the meta-level methods associated with normal objects. The interpreter can assume that it will never need to reify them because the object's implicit mirror cannot be changed. Nevertheless, one could allow a `becomeMirroredBy:` operation on mirages only. However, simply assigning the implicit mirror of a mirage may break the strict one-to-one relationship established between the mirage and its unique implicit mirror: if a mirage `m`'s implicit mirror `m1` would be replaced by an implicit mirror `m2`, then `m2.base` may not properly refer back to `m` but to another mirage. To avoid such issues, we opted to keep the one-to-one relationship constant for objects.

7. Related and Future Work

We now briefly discuss how the present architecture differs from the previous version of AmbientTalk [4], as well as related work in the area of implicit 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 features specifically for mobile ad hoc networks [4]. 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 actor's 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 its reserved name.

In AmbientTalk/1 reflection is neither stratified nor encapsulated: base-level code can be affected by the implementation details of metalevel code such as that of a new language feature. Because the base- and metalevels are not partitioned into separate namespaces, name clashes between the two levels can 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. Finally, AmbientTalk/1 had no metaobject protocol for objects, only for actors.

7.2. Related Work

Intercession – the ability of a program to modify its own execution semantics– has been present in early work on reflection in general [1] and reflection in object-oriented languages in particular [2]. Since then, there have been numerous proposals to introduce 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 intercession –such as interception of message sending, object creation, etc.– from the start. A notable exception is the CLOS MOP [3, 23], 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]. Moreover, CLOS has no equivalent for mirrors on actors, because it is not a language based on event loop actors.

Because the interception of messages sent to objects is a common use case of 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 [21], such as method wrappers [24] or using the `doesNotUnderstand:` protocol. 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 `doesNotUnderstand:`, 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 5.1.1, AmbientTalk's stratified MOP avoids such name clashes.

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 intercession rely on proxies. The dynamic proxies added to Java 1.3 [25] do introduce a form of stratification: using our terminology, the proxy is an empty mirage object while the associated `InvocationHandler` acts as its (implicit) mirror. However, with respect to encapsulation, there is no equivalent of a mirror factory to access the `InvocationHandler` of a given proxy.

Bytecode transformation is another technique for intervening in the method lookup process of a language [26, 27]. Recently, techniques relying on bytecode transformation have been used to add fine-grained implicit reflection to Smalltalk [28, 29]. On the one hand, these transformation-based approaches mostly ignore the principles of mirror-based architectures, especially structural correspondence: introspecting 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, such as the ones of Self and Strongtalk, offer limited intercession [5]. The architecture presented in this paper precisely reconciles mirrors with intercession.

The combination of reflection and actors was an active research topic in the late eighties/early nineties. Our support for reflection on actors has been inspired by early reflective actor languages such as ABCL/R [30] and AL-1/D [31]. The innovation of AmbientTalk's reflection on actors is that the actors reflected upon are event loops rather than "active objects", as they have been traditionally represented. AmbientTalk is the first language to combine an E-style concurrency model [15] with an advanced reflective architecture. The message passing protocol that transmits asynchronous messages from one actor to another has been largely inspired by the protocol introduced in CoDA [32].

Our work also relates to partial behavioral reflection [33]: the principle of limiting the cost of implicit reflection to where and when it is really needed. The idea is to only intercess (i.e. reify a certain metalevel operation) at those execution points where a reification is really necessary. AmbientTalk's implicit mirrors currently support what is known as *entity selection*: the ability to activate intercession on a per-object basis. That is: the AmbientTalk interpreter knows that it only needs to reify metalevel operations performed on *mirage* objects. Non-mirage objects follow the default language semantics

and thus require no intercession. There exist more fine-grained selection levels, such as *operation* and *intra-operation selection* [33]. These features are particularly useful for efficiently supporting aspect-oriented extensions [34, 33], and can be provided by the language processor [35]. Incorporating them in the AmbientTalk interpreter is future work. A benchmark of early experiments in adding operation selection indicates promising gains in performance [7].

7.3. Current Status and Future Work

An open-source interpreter for the AmbientTalk language has been implemented in Java and, as noted previously, can be downloaded from `http://prog.vub.ac.be/amop`. In addition to the desktop, the implementation runs on the Java 2 micro edition (J2ME) platform, under the connected device configuration (CDC). Hence, AmbientTalk runs on PDAs and high-end cellular phones. Our current experimental setup consists of a number of smartphones which communicate by means of an ad hoc WLAN network. The reflectively implemented future-type message passing and leased far reference language features are shipped with AmbientTalk's standard library. Their actual implementations are more elaborate than the didactic versions described in this paper, but in essence they are equivalent.

Partial Behavioral Reflection In future work, we would like to investigate whether the performance gain achieved by incorporating partial behavioral reflection in the mirror-based architecture outweighs its associated complexity cost. One possibility to strike a balance between both is to give the "protocols" which group logically related meta-operations (cf. sections 4.1 and 4.2) a first-class status as separate objects. For example, rather than having `invoke` as a method defined on an object's mirror `m` directly, the method would be defined on the protocol object `m.invocationProtocol`. Having partitioned all methods into separate protocol objects, the next step would be to make protocol objects replaceable. For example, rather than overriding `invoke`, one replaces the mirror's `invocationProtocol` object with a new protocol object (with its own definition of `invoke`). Replacing only a single protocol object and leaving the others intact leads to more fine-grained partial behavioral reflection, because the implementation of metalevel operations can be hard-wired for protocol objects that have not (yet) been replaced.

Security AmbientTalk is a reflective distributed programming language, which raises the issue of how objects can be secured against malicious programs. AmbientTalk's security model is based on the object-capability model as embodied in the E language [36]. However, in E the trust boundary is defined at the individual object level. In AmbientTalk, we currently assume (but not yet enforce) that objects local to the same actor can be trusted. The trust boundary is thus defined at the actor level. From the point of view of the reflective architecture, the question then becomes what capabilities a metaprogrammer can exert on a remote object. In AmbientTalk, a mirror on a far reference currently does *not* give the metaprogrammer more rights at the meta-level (e.g. inspecting or reading the slots of a remote object), except for the ability to reflectively send an asynchronous message across the far reference. Thus, remote objects cannot by default be inspected or taken apart. However, remote reflection on an object is still possible if the inspector is given a far reference *to the mirror* of the remote reflectee (which is distinct from a far reference *to the reflectee itself*). In this case, the far reference to the mirror must be explicitly parameter-passed to the inspector by a party trusted by the reflectee, following the usual rules of object-capability security.

8. Conclusions

We have introduced the mirror-based metalevel architecture of the distributed programming language AmbientTalk. AmbientTalk programs consist of concurrently executing actors each encapsulating one or more objects. Metaprograms can inspect actors and their objects by means of mirrors, which are metaobjects created by a separate mirror factory. While mirrors traditionally provide good support for introspection and self-modification, they lack support for intercession. Furthermore, to the best of our knowledge, mirror-based reflection has not previously been applied to an event loop actor-based language.

AmbientTalk reconciles traditional, introspective mirrors with intercession by dinstinguishing between *explicit* and *implicit* mirrors. Implicit mirrors are used by the interpreter itself when performing metalevel operations on base-level objects. In order to causally connect such mirrors to base-level objects, AmbientTalk introduces *mirages*: objects whose MOP is implemented by a causally connected implicit mirror. AmbientTalk also supports implicit mirrors on actors as a whole.

Because AmbientTalk successfully combines mirrors with implicit reflection, it can introduce the benefits of mirror-based reflection in the implementation of language features which require intercession. First, to metaprograms, mirrors remain only accessible via the mirror factory, allowing an object to encapsulate the metalevel behavior implementing the new language feature. Second, implicit mirrors are stratified with respect to base-level code, such that extensions to the metaobject protocol do not interfere with application-level code. We have illustrated these benefits by implementing two non-trivial language extensions using implicit mirrors, namely future-type message passing and leased far references.

REFERENCES

1. Smith BC. Reflection and semantics in Lisp. *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, ACM Press: New York, NY, USA, 1984; 23–35.
2. Maes P. Concepts and experiments in computational reflection. *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, ACM Press: New York, NY, USA, 1987; 147–155.
3. Kiczales G, Rivieres JD, Bobrow DG. *The Art of the Metaobject Protocol*. MIT Press: Cambridge, MA, USA, 1991.
4. Dedecker J, Van Cutsem T, Mostinckx S, D'Hondt T, De Meuter W. Ambient-oriented Programming in Ambienttalk. *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, *Lecture Notes in Computer Science*, vol. 4067, Thomas D (ed.), Springer, 2006; 230–254, doi:10.1007/11785477_16.
5. Bracha G, Ungar D. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2004; 331–343.
6. Agesen O, Bak L, Chambers C, Chang BW, Hölszle U, Maloney J, Smith R, Ungar D, Wolczko M. The SELF 4.1 programmer's reference manual. *Technical Report*, Sun Microsystems, Inc. and Stanford University 2000.
7. Mostinckx S, Van Cutsem T, Timbermont S, Tanter E. Mirages: Behavioral intercession in a mirror-based architecture. *Proceedings of the Dynamic Languages Symposium (DLS)*, Costanza P, Hirshfeld R (eds.), ACM Press, 2007; 222–248.
8. Bracha G, Griswold D. Strongtalk: Typechecking Smalltalk in a Production Environment. *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, 1993; 215–230.
9. Ferber J. Computational reflection in class based object-oriented languages. *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, ACM: New York, NY, USA, 1989; 317–326, doi: http://doi.acm.org/10.1145/74877.74910.
10. Van Cutsem T, Mostinckx S, Gonzalez Boix E, Dedecker J, De Meuter W. AmbientTalk: object-oriented event-driven programming in mobile ad hoc networks. *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, IEEE Computer Society, 2007; 3–12, doi:10.1109/SCCC.2007.12.

11. Ungar D, Smith RB. Self: The power of simplicity. *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, ACM Press, 1987; 227–242.
12. Lieberman H. Using prototypical objects to implement shared behavior in object-oriented systems. *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, ACM Press, 1986; 214–223.
13. Agha G. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
14. Yonezawa A, Briot JP, Shibayama E. Object-oriented concurrent programming in ABCL/1. *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, 1986; 258–268.
15. Miller M, Tribble ED, Shapiro J. Concurrency among strangers: Programming in E as plan coordination. *Symposium on Trustworthy Global Computing*, *LNCS*, vol. 3705, Nicola RD, Sangiorgi D (eds.), Springer, 2005; 195–229.
16. Mascolo C, Capra L, Emmerich W. Mobile Computing Middleware. *Advanced lectures on networking*. Springer-Verlag New York, Inc., 2002; 20–58.
17. Maes P, Nardi D ((eds.)). *Meta-Level Architectures and Reflection*. North-Holland: Alghero, Sardinia, 1988.
18. Guy L Steele J. Growing a language. *Higher Order Symbol. Comput.* 1999; **12**(3):221–236, doi:http://dx.doi.org/10.1023/A:1010085415024.
19. Gray C, Cheriton D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, ACM Press: New York, NY, USA, 1989; 202–210, doi:http://doi.acm.org/10.1145/74850.74870.
20. Caromel D. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming* Nov 1989; **2**(4):12–18.
21. Ducasse S. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming* 1999; **12**(6):39–44.
22. Mulet P, Malenfant J, Cointe P. Towards a methodology for explicit composition of metaobjects. *Proceedings of the 10th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, ACM Press: Austin, Texas, USA, 1995; 316–330. ACM SIGPLAN Notices, 30(10).
23. Paepcke A. User-level language crafting: Introducing the CLOS metaobject protocol. *Object-oriented programming: the CLOS perspective*. MIT Press: Cambridge, MA, USA, 1993; 65–99.
24. Brant J, Foote B, Johnson R, Roberts D. Wrappers to the rescue. *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP 98)*, *Lecture Notes in Computer Science*, vol. 1445, Springer-Verlag: Brussels, Belgium, 1998; 396–417.
25. Sun Microsystems. Dynamic Proxy Classes 1999. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`, accessed August 29th, 2008.
26. Chiba S, Nishizawa M. An easy-to-use toolkit for efficient Java bytecode translators. *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2003)*, *Lecture Notes in Computer Science*, vol. 2830, Pfenning F, Smaragdakis Y (eds.), Springer-Verlag: Erfurt, Germany, 2003; 364–376.
27. Welch I, Stroud RJ. Kava - using bytecode rewriting to add behavioral reflection to Java. *Proceedings of USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, San Antonio, Texas, USA, 2001; 119–130.
28. Denker M, Ducasse S, Tanter É. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures* Jul 2006; **32**(2-3):125–139.
29. Röthlisberger D, Denker M, Tanter É. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures* July-October 2008; **34**(2-3):46–65.
30. Watanabe T, Yonezawa A. Reflection in an object-oriented concurrent language. *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, 1988; 306–315.
31. Okamura H, Ishikawa Y, Tokoro M. AL-1/D: A distributed programming system with multi-model reflection framework. *Proceedings of the Workshop on New Models for Software Architecture*, 1992.
32. McAffer J. Meta-level programming with coda. *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, Springer-Verlag: London, UK, 1995; 190–214.
33. Tanter É, Noyé J, Caromel D, Cointe P. Partial behavioral reflection: Spatial and temporal selection of reification. *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, ACM Press: Anaheim, CA, USA, 2003; 27–46. ACM SIGPLAN Notices, 38(11).
34. Hilsdale E, Hugunin J. Advice weaving in AspectJ. Lieberherr [37]; 26–35.
35. Bockish C, Haupt M, Mezini M, Ostermann K. Virtual machine support for dynamic join points. Lieberherr [37]; 83–92.
36. Miller M, Morningstar C, Frantz B. Capability-based financial instruments. *Proceedings of Financial Cryptography*, springer-Verlag, 2000.
37. Lieberherr K ((ed.)). *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, ACM Press: Lancaster, UK, 2004.

Table I. API of Explicit Mirrors on Objects.

Structural access protocol	
`addSlot(slot)`	Adds a field or a method slot to the object.
`removeSlot(selector)`	Removes an owned slot from the object.
`grabSlot(selector)`	Returns an owned slot whose name matches the selector.
`listSlots()`	Returns an array of all owned slots.
`select(delegate,selector)`	Returns a closure which, when applied, invokes the slot corresponding to the selector on the mirrored object.
Method invocation protocol	
`invoke(delegate, invocation)`	Invokes a method (with **`self`** bound to `delegate`).
`receive(message)`	Makes the object receive an asynchronous message.
`send(receiver,message)`	Makes the object send an asynchronous message.
`respondsTo(selector)`	Asks the object whether or not it (or one of its parents) has a slot matching the selector.
Object instantiation protocol	
`clone()`	Creates a shallow copy of the object (except for the object's **`super`** slot, whose value is recursively cloned as well).
`newInstance(arguments)`	Clones the object and invokes `init` on the clone.
Type tag protocol	
`isTaggedAs(typeTag)`	Asks the object whether it or one of its parents is tagged as a subtype of the given type tag.
`listTypeTags()`	Returns an array of type tags with which the object is tagged.

Table II. API of Explicit Mirrors on Actors.

Message sending protocol	
`send(receiver,message)`	Sends a message asynchronously to the receiver.
Message reception protocol	
`schedule(receiver,message)`	Adds a letter containing the message and the receiver to the mailbox.
`listIncomingLetters()`	Returns an array of all letters currently in the mailbox.
Service discovery protocol	
`listPublications()`	Returns an array of advertised local objects.
`listSubscriptions()`	Returns an array of active discovery event handlers.
`publish(object,typeTag)`	Advertises a local object as a discoverable service.
`subscribe(typeTag,closure)`	Registers a closure as a discovery event handler.

Table III. Additional methods in the API of Implicit Mirrors on Objects.

Method invocation protocol	
`doesNotUnderstand(selector)`	Invoked when the object receives a message for which it has no matching slot.
Object marshalling protocol	
`pass()`	Invoked when an object is marshalled. Returns the object to be marshalled instead of this object.
`resolve()`	Invoked when an object is unmarshalled. Returns the object replacing the unmarshalled object.

Table IV. Additional methods in the API of Implicit Mirrors on Actors.

Message sending protocol	
`createMessage(name,args,tags)`	Creates a message from name, arguments and type tag annotations.
Message reception protocol	
`serve()`	Dequeues a letter from the mailbox and process it.
Mirror creation protocol	
`createMirror(onObject)`	Returns an explicit mirror on a local object.
`getExplicitActorMirror()`	Returns an explicit mirror on the actor.
Reference creation protocol	
`createReference(toObject)`	Returns a far reference to a local object.

Table V. Reflective Operations to Retrieve and Install Mirrors.

Reflecting on Objects	
`reflect:` `obj`	Returns an explicit mirror reflecting on the given object.
`object:` `occ` **`mirroredBy:`** `mcc`	Creates a mirage, with the object returned by the `mcc` closure as its implicit mirror.
Reflecting on Actors	
`reflectOnActor``()`	Returns an explicit mirror on the local actor.
`becomeMirroredBy:` `actorMirror`	Installs a new implicit actor mirror.