Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
Programming Technology Laboratory

A Reflective Approach to Building Extensible Distributed Actor-based Languages

Dissertation submitted in partial fulfillment of the requirements for the degree of
Licentiaat in de Toegepaste Informatica

Pierre Martin

Academic Year 2005-2006

Promotor:	Prof. Dr. Theo D'Hondt
Advisor:	Tom Van Cutsem

May 2006

Dedicated to the memory of Pierre Maes, my grandfather (1923-2006)

Abstract

With the democratization of wireless devices and the mobile networks they imply, the need for a new programming paradigm especially geared towards the development of distributed applications for such networks is becoming obvious. The ambient-oriented programming paradigm is an attempt at incorporating the characteristics of such mobile networks at the heart of a new computational model. Being a fledgling paradigm, one wants to experiment with new language constructs in the context of ambient-oriented environments. One way to achieve this is to use a technique called computational reflection. Computational reflection allows one to tailor a programming language to his specific needs. AmbientTalk is one such reflective language, built specifically to act as a "language laboratory" for exploring the ambient-oriented programming paradigm. However, its reflective architecture exhibits multiple drawbacks.

This dissertation attempts to resolve these drawbacks. To this end, it analyzes the reflective design principles already put forward by researchers in the field of reflection and meta-level architectures, and assesses their applicability in the context of mobile networks. AmbientTalk's reflective architecture is subsequently enhanced following the most applicable design principles. The resulting reflective architecture is based on general properties, making it applicable, mutatis mutandis, to other distributed actor-based languages as well. AmbientTalk's enhanced reflective architecture is extensively validated by reimplementing existing ambient-oriented language constructs and by observing their improved characteristics.

Samenvatting

Omwille van de democratisering van draadloze toestellen en de mobiele netwerken die er uit voortvloeien ontstaat de nood aan een nieuw programmeerparadigma dat specifiek gericht is op de ontwikkeling van gedistribueerde toepassingen. Het *ambient-geöriënteerd* programmeerparadigma is een poging om de karakteristieken van zulke netwerken op te nemen aan de grondslag van een nieuw computationeel model. Om dit paradigma tot volwassenheid te brengen dient er geëxperimenteerd te worden met nieuwe taalconstructies in de context van ambient-geöriënteerde omgevingen. Een manier om dit te bewerkstelligen is het gebruik van een techniek genaamd *computationele reflectie*. Computationele reflectie laat de programmeur toe om een programmeertaal aan te passen aan zijn specifieke behoeften. AmbientTalk is een dergelijke reflectieve taal die specifiek ontworpen is als een "taallaboratorium" om het ambient-geöriënteerd programmeerparadigma te onderzoeken. De reflectieve architectuur van AmbientTalk vertoont echter verscheidene gebreken.

Deze verhandeling probeert die gebreken aan te pakken. Om dit te bereiken onderzoekt het de verschillende ontwerpprincipes voor reflectieve architecturen die reeds door onderzoekers naar voren werden geschoven, en evalueert het hun toepasbaarheid in de context van mobiele netwerken. Vervolgens wordt de reflectieve architectuur van AmbientTalk verbeterd met de meest toepasbare ontwerpprincipes. Het resultaat is een reflectieve architectuur die gebaseerd is op algemene eigenschappen, en, mutatis mutandis, ook toegepast kan worden op andere gedistribueerde *actor-gebaseerde* talen. De verbeterde reflectieve architectuur van AmbientTalk wordt uitvoerig gevalideerd door bestaande ambient-geöriënteerde taalconstructies te herimplementeren en door hun verbeterde eigenschappen waar te nemen.

Acknowledgements

I would like to express my gratitude towards all the people who supported me (in very diverging ways) during the writing of this dissertation:

Prof. Dr. Theo D'Hondt for promoting my research.

Tom Van Cutsem for guiding me through every step of the writing process, and for never renouncing to help me. Words can barely express how grateful I am towards him for his continuous help and encouragements, especially when I was thinking that I would never be able to finish this dissertation.

My parents for letting me study what I want at the excellent facilities of the Vrije Universiteit Brussel.

Jane, Gilles and Patrick for their moral support and for just being who they are.

Everybody at PROG for their feedback during my presentations.

The fine people at Infogroep for their friendliness and for constantly reminding me that I would never finish this dissertation in time.

Contents

List of Figures

1 Introduction

1.1 Research context

The last years have seen a swift emergence of wireless devices of various sizes and computational power. This is mainly due to the democratization of such devices and to the freedom they proffer to their users, who can carry them around freely and almost instantly join other existing mobile networks implied by surrounding devices. These mobile networks are constituted of a set of possibly very contrasting wireless devices, such as cellular phones, PDAs, printers, "intelligent" watches, ... that are all in a certain range of each other and communicate by means of well-defined protocols (e.g. Wi-fi and Bluetooth). Because of their omnipresence around the user, the wireless devices in these mobile networks are sometimes also said to form a PAN (Personal Area Network) [BGH00].

The freedom that wireless devices involve does, however, come at a cost when one wants to develop applications to make them interact. As a matter of fact, mobile networks differ significantly from traditional "stationary" networks. For instance, mobile networks are characterized by a limited communication range, due to limitations of the underlying hardware. In practice, this leads to more frequent disconnections as devices may move out of one another's communication range unannounced. Compared to stationary networks, one can state that disconnections are becoming more of a rule rather than an exception in mobile networks. In the same vein, wireless devices cannot rely on the explicit knowledge of the availability and location of other wireless devices, because of the strong mobility and high failure factor in mobile networks. Hence, whereas devices in stationary networks can rely on URLs to address other stationary devices, wireless devices often depend on a *service discovery protocol* to discover peers. These issues, along with other characteristics of mobile networks, are discussed in more detail in chapter 3.

To tackle these technical difficulties and to offer the application programmer suitable abstractions to deal with disconnections and service discovery, middleware solutions were developed for existing programming languages. Examples of such middleware solutions include *data-sharing oriented middleware*, which tries to maximize the autonomy of temporarily disconnected devices using weak replica management, and *tuple space-based middleware*, which uses a global tuple space in which processes can asynchronously publish and query tuples. However, none of these approaches addresses all problems associated with mobile networks. For this reason, a new paradigm, called the *ambient-oriented programming paradigm*, was developed [DVM+06]. The ambient-oriented paradigm tries to extend the object-oriented paradigm in such a way that the characteristics of mobile networks are incorporated at the very heart of the computational model.

In order to experiment with ambient-oriented environments, the *AmbientTalk* language was designed at the *Programming Technology Laboratory* of the *Vrije Universiteit Brussel*. It is based on the ambient actor model [DB04], a model of distribution in which applications consist of a set of local and remote actors (i.e., in essence, active objects with their own thread of execution and a number of message queues for communication), which communicate with each other through asynchronous message passing. AmbientTalk encompasses all the characteristics of the ambient-oriented paradigm, and is implemented in a such a way that it can easily be extended with new behaviour. This feature is extensively used to experiment with new language constructs in the context of ambient-oriented programming [DVM+06].

1.2 Objectives

As noted in the previous section, AmbientTalk features a language kernel that is easily extensible with new behaviour. The underlying technique used to support this high extensibility is called *computational reflection* [Mae87]. In a nutshell, computational reflection allows a language to reflect on (and possibly intervene in) its own computation by opening up (parts of) its implementation to the programmer through a well-defined interface.

Computational reflection was first introduced in a procedural programming language [Smi82], but has since then evolved to be present in many languages implementing equally many language paradigms. For instance, reflective object-oriented languages typically support reflection by means of special *meta-level* objects, which are used to reflect upon *base-level* ("normal") objects, as we know them from object-oriented languages. Such meta-objects are typically bound to one another by formally defined *protocols* that describe how they interact to achieve certains results. Knowledge of such protocols—the sum of which is called the *meta-object protocol* of a language—allows the programmer to write powerful meta-programs to e.g. dynamically adjust the behaviour of the language at runtime.

However, meta-object protocols in their most basic form do not address the issues of concurrency and distribution—two key concepts in ambient-oriented programming languages. Languages like ABCL/R [WY88], based on the functional actor model, address the former issue by introducing a meta-architecture for concurrently running active objects. They do nonetheless fall short when it comes to deploying them in highly volatile mobile networks. For example, ABCL/R does not feature a service discovery protocol, but instead relies on the explicit knowledge of the availability and location of other actors.

As a consequence, the objectives of this dissertation are twofold. Firstly, we seek to gain insight in the requirements for meta-level architectures for ambient-oriented programming languages based on the ambient actor model. For example, a proper synchronization mechanism between the base-level and the meta-level of a distributed reflective language is one of the requirements that readily comes to mind.

Secondly, we attempt to adjust the current minimal meta-object protocol of AmbientTalk to comply with these requirements by applying to it a number of solid design principles for meta-level architectures. AmbientTalk's meta-object protocol is used extensively as a *language laboratory* for experimenting with new ambient-oriented language constructs [DVM$^+$06]. By applying these design principles to its meta-object protocol, we seek to augment its power as a language laboratory by ensuring that language constructs become more separable from the base-level, more encapsulated and more easily composable with one another.

1.3 Methodology

As stated in the previous section, the AmbientTalk language forms the starting point of our research, partly because of its suitability to be deployed in highly volatile mobile networks. As we shall see, AmbientTalk does already feature a minimal meta-object protocol. It does, however, present various limitations, the most important of which probably being the complete lack of separation between the base-level and meta-level of the language. We will therefore scrutinize AmbientTalk's exising meta-object protocol, and point out all its limitations and disadvantages. Our goal is to correct these disadvantages by applying sound design principles already put forward by researchers in the field of reflection and meta-level architectures. As such, the resulting meta-object protocol is based on general properties, making it applicable not only to AmbientTalk but allowing it to be applied to other actor-based languages as well.

The validity of our changes to the meta-object protocol of AmbientTalk is tested by reimplementing existing ambient-oriented language constructs and by observing their improved characteristics. Throughout this disseration we will therefore repeatedly use two such constructs to support our case, namely non-blocking futures (a technique that reconciles asynchronous communication with classic synchronous method call return values) and due-blocks (a construct that allows one to put time constraints on outgoing asynchronous messages).

1.4 Contributions

The contributions of this dissertation are twofold with respect to the current research in meta-level architectures for distributed actor-based languages. Firstly, it provides an evaluation of the current state of the art in computational reflection for distributed actor-based languages, focusing on the ambient-oriented aspects described in section 1.1. It also evaluates the applicability of typical object-oriented meta-level constructs and design principles in the context of distributed languages based on the ambient actor model.

Secondly, it uses the lessons learned from this in-depth evaluation to design and implement a mirror-based meta-object protocol (i.e. a meta-object protocol that respects a set of strict design principles, which will be introduced in chapter 2) for AmbientTalk. Ultimately, the goal of the enhanced mirror-based meta-architecture is to experiment with new language constructs. We will extensively show how the architecture proposed in this dissertation is more geared towards this purpose than AmbientTalk's current meta-object protocol.

1.5 Dissertation roadmap

This dissertation starts with an overview of computational reflection in object-oriented and actor-based languages. The goal is to establish the necessary background on the state of the art in object-oriented reflection, such that we can reuse this knowledge subsequently to implement a proper meta-object protocol for a distributed language based on the ambient actor model.

Chapter 3 introduces mobile networks and their main characteristics. It uses these characteristics to derive the foundations of the ambient-oriented programming paradigm. Subsequently, AmbientTalk is introduced as our experimental platform. Particular attention is paid to its existing reflective architecture, which is also illustrated in detail using two language constructs: non-blocking futures and due-blocks. Chapters 2 and 3 can be read in parallel.

Chapter 4 evaluates the shortcomings of AmbientTalk's existing meta-object protocol, namely the lack of *stratification* (i.e. separation between the base-level and the meta-level) and *encapsulation*, and the absence of a high-level mechanism to combine language constructs together. It also evaluates the concepts presented in chapter 2 in the light of implementing a proper meta-object protocol for AmbientTalk.

Chapter 5 aims to tackle the first lacuna of AmbientTalk's meta-object protocol identified in chapter 4, by describing the design and implementation of a new meta-object protocol for AmbientTalk based on the principle of mirrors, as described in chapter 2. The mirror-based implementation is validated by rewriting the two language constructs presented in chapter 3 to comply with the new meta-object protocol.

The second issue associated with AmbientTalk's existing reflective architecture is for its part addressed in chapter 6, which explains how the mirror-based meta-object protocol presented in chapter 5 can be extended to ensure that language constructs such as e.g. non-blocking futures and due-blocks are combined correctly.

This dissertation is concluded in chapter 7, which briefly returns to our achievements and also assesses their limitations, leaving room for future work and giving some pointers for future research in the continuity of this field. Lastly, appendices A and B are included for the interested reader who wishes to explore the implementation details of the designs presented in chapters 5 and 6.

2 Reflection in Object-Oriented Programming Languages

This chapter introduces the concept of computational reflection as a mechanism to build extensible and customizable programming languages. It concentrates mainly on reflection in object-oriented languages, as this forms the basis for our research. Meta-object protocols are introduced as a mechanism to adapt the behaviour of an object-oriented language to one's particular needs. Subsequently, the notion of mirrors is introduced. Mirros aim to clearly separate the base-level from the meta-level in reflective object-oriented languages. Lastly, the implications of concurrency in a reflective object-oriented language are assessed. Every design principle is illustrated with a case study of an existing reflective language architecture. This includes CLOS (a class-based object-oriented language), Self (a prototype-based object-oriented language) and ABCL/R (an object-oriented concurrent language).

2.1 Computational reflection

Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation [Mae87]. Whereas conventional programs manipulate data from an application-specific problem domain (e.g. a program that computes the fixpoint of a mathematical function or a program that renders a three-dimensional drawing), a reflective computation manipulates data that represents the state and the structure of the program itself. Hence, the problem domain of a reflective computation is the program itself. The former is called a *base-program*, while the latter is a *meta-program* that manipulates its own computation. Note that not all meta-programs are reflective computations. For example, an interpreter for a language L written in a language L' is also a meta-level program: language L' is used to perform computations about language L. However, by doing so, language L does not (necessarily) perform computations about itself.

To reason about computations, meta-programs need a representation of the structures (e.g. entities, relations, ...) of the computation. In our language interpreter example, suppose L' is an object-oriented language and L is a procedural language, then L' may for example contain a `Procedure` class to represent procedures of L. In a reflective computation, such representations symbolize their own computation, and are called *reifications* of the computation. Reifications differ from representations in that they are *causally connected* to the domain they represent. This means that if either the reification or the domain it *reifies* (i.e. represents in a causally connected way) changes, this leads to a corresponding effect upon the other [Mae87], or, simply put, that changes at the meta-level are reflected at the base-level and vice versa. For example, if the programmer adds a slot to an existing base-level object using a meta-level operation, the added slot must immediately be visible at the base-level. Conversely, meta-level operations must always accurately represent the base-level entities.

To sum up, a reflective system is a system that incorporates structures representing (aspects of) itself. The sum of these structures is called the *self-representation* of the system [Mae87]. This self-representation allows the system to answer questions about itself and act upon itself. Because the self-representation is causally connected to the aspects of the system it represents, the following holds:

1. The system always has an accurate representation of itself.
2. The status and computation of the system are always in compliance with this representation. This means that a reflective system can bring modifications to itself as a result of its own computation.

2.1.1 What to reify

The nature of what is being reified by a reflective language depends on the programming paradigm adopted by the language. For example, a rule-based logic programming language might reify its rule inference mechanism, thereby allowing the progammer to explicitly invoke it or possibly even to override it with custom behaviour. An object-oriented language, on the other hand, might reify the object instantiation and message sending mechanisms, as well as the structure of objects (e.g. to allow the programmer to query an object for its data slots).

2.1.2 How much to reify

Reflective language architectures are often characterized in terms of their support for [KP94]:

1. **Introspection.** Introspection is defined as the ability of a program to examine its own structure. For example, a reflective object-oriented language architecture might provide a mechanism for returning a list of all slots of an object, a representation of the class of an object, of the source code of method bodies, ...

2. **Explicit invocation.** A reflective language architecture supports explicit invocation if it allows the programmer to explicitly invoke substrate operations, i.e. operations that are typically only called internally by the language interpreter. Examples include the `eval` and `apply` operations of a reflective functional language, or an object instantiation operation like `new` in an object-oriented language.

3. **Intercession.** Intercession is the ability to modify the semantics of the underlying programming language from within the language itself, e.g. to add/modify state (for example, to add a new method to an already existing class) or to add/modify meta-level behaviour (for example, to change the way method lookup and object initialization is performed). The former is referred to as *structural intercession* and the latter as *behavioural intercession*.

Examples of reflective languages include CLOS, which supports introspection, explicit invocation and intercession (see section 2.2.1), and Java, which supports introspection and explicit invocation in a limited way.

2.1.3 How to reify

The form in which base-level entities are reified often depends on the programming paradigm adopted by the language. For example, some logic-based languages with a reflective architecture like FOL [Wey78] adopt the concept of a meta-theory. A meta-theory is similar to a (base-)theory except that it is used to reason about other theories, instead of about the external problem domain. Examples of predicates used in a meta-theory are `provable(Theory, Goal)` and `clause(Left-hand, Right-hand)` [Mae87]. In a reflective functional language, evaluation and function application might be reified as an `eval` and an `apply` *function*, respectively.

An object-oriented language architecture, on the other hand, might use meta-level objects and methods to reify (parts of) its objects' internals as well as internal interpreter operations. This approach to building a reflective interface that reifies both the structure and the behaviour of an object-oriented computation is called a *meta-object protocol* [KdRB91], and is discussed in more detail in section 2.2.

2.1.4 Applications of reflection

Reflection is used widely for a myriad of applications. For example, the introspective capabilities of a language are typically used to create class browsers and code inspectors. The *Java Core Reflection* can, for example, be used to implement a class browser [BU04]. Structural intercession can for its part be used to create debugging tools. An example of this is the *Java Debug Interface* (JDI) [Mic], which can be used to debug local as well as remote Java applications [BU04].

Behavioural intercession is perhaps the most powerful form of reflection, and has many diverging uses. In CLOS, it can amongst others be used to implement object persistency (see section 2.2.1). Tracing method calls and profiling code also belong to the possibilities offered by behavioural intercession (the following section gives an example of a simple method tracer). Additionally, reflection also has applications in the field of language extensions (as we shall see in chapter 3, the reflective capabilities of the distributed language AmbientTalk are used extensively to experiment with new language extensions). In concurrent environments, reflection has also been proved successful at optimizing certain programs. For example, the n-Queens parallel search problem can be optimized at the meta-level using *locality control*, so that child nodes at deep level in the search tree are created at the same processor as their parents' in order to reduce remote communication overhead [MMAY95].

2.2 Meta-object protocols

The idea of using *meta-objects* (i.e. objects that reside at the meta-level) to reify object-oriented entities was first introduced by Maes [Mae87]. In her experimental language 3-KRS, every entity (e.g. instances, slots, methods, messages, ...) is an object. Furthermore, there exists a one to one relation between every such object and the meta-object that describes the object's meta-level behaviour. For example, an object representing a person might contain methods to calculate its age, while its associated meta-object would contain methods that handle incoming messages, object instantiation, method lookup through the inheritance chain, etc.

Furthermore, the way in which meta-objects interact with each other can be formalized in a protocol. Hence, this *meta-object protocol* describes the responsibility of each meta-object in the architecture [KdRB91]. For example, a protocol might formally define how every message send from a `sender` base-object to a `receiver` base-object is handled. Such a *message send* protocol might for instance be defined as follows:

```
` MESSAGE SEND PROTOCOL `
` defines how "receiver.msg(args)" from within the context `
`  of a "sender" object is executed                        `
1. call send(receiver, "msg", args) method of sender's meta-object
```

```
2. call receive(sender, "msg", args) method of receiver's meta-object
3. call process("msg", args) method of receiver's meta-object
```

Additionally, the splitting of base- and meta-level objects allows the programmer to temporarily assign another meta-object (possibly a specialized child object of the original meta-object) to a base-level object. This, together with well-defined responsibilities between meta-objects, allows the programmer to dynamically adapt the language architecture to his specific needs. As a matter of fact, the programmer who is aware of the meta-object protocol of the language can intervene in every step of the protocol.

As a concrete example, consider a programmer who wants to trace every message received by an object by printing an informative line of output. In a non-reflective language architecture, there are basically two ways to achieve the desired result: the programmer can either extend every method with `print` statements, or recompile the language interpreter after adding the necessary output printing code to the native message reception routine. Clearly, none of both approaches are satisfactory. On the other hand, suppose that we have a reflective language architecture with a meta-object protocol where message sends are performed as described above. In this case, it suffices to temporarily replace the meta-object of the receiving object with a child object of its default meta-object in which the `receive` method is overridden:

```
receive(sender, msgName, argList) {
  print("Received '", msgName, "' from '", sender.toString(),
        "with arguments ", argList.toString());
  super.receive(sender, msgName, argList);
}
```

Hence, the code of both the interpreter and the base-level object's methods remains unchanged, and only the meta-level objects are changed.

2.2.1 Case study: The CLOS meta-object protocol

Common LISP Object System (CLOS) [BDG$^+$88] is an object-oriented extension of LISP which features an extensive meta-object protocol (MOP) [KdRB91]. This section describes parts of CLOS's MOP and gives a concrete example in which the MOP is used to transparently implement persistent objects (i.e. objects whose state is stored in a database and retrieved when needed). The full implementation can be found in literature [Pae93]. We begin with a short overview of the base-level of the language.

2.2.1.1 Base-level

CLOS is a class-based object-oriented extension to LISP. This means that objects are instantiated from classes. Classes are created using `defclass`, while instances are created using `make-instance`, like so:

```
(defclass Person () ; list of superclasses
                  (; "name" slot, accessible through the automatically
                   ; created "getName" accessor
                   (name :accessor getName
                         :initarg :name)
                   ; "age" slot whose default value is 0
                   (age :initform 0
                        :initarg :age
                        :accessor getAge)))
(defclass Teacher (Person) ; Teacher inherits from Person
                  ((subjects :accessor getSubjects
                             :initform '())))

; bind instances to variables
(setq john (make-instance 'Person :name "John" :age 20))
(setq doe (make-instance 'Teacher :name "Doe" :age 30))
```

This defines a `Person` class with two slots: `name` and `age`. The second `defclass` statement defines a `Teacher` class that inherits from `Person`. In the example above, instances of both classes are created using `make-instance`.

Methods are defined in CLOS using `defmethod`. Every parameter of a method can be specialized to a particular class, by specifying an optional *specializer*. The receiver has to be specified explicitly. The example below illustrates method definition:

```
; newSubject can be an instance of any class
(defmethod addSubject ((self Teacher) newSubject)
  (setf (getSubjects self) (cons newSubject (getSubjects self))))
; nYears must be an instance of the "integer" class
(defmethod increaseAgeBy ((self Person) (nYears integer))
  (setf (getAge self) (+ (getAge self) nYears)))
```

Armed with this knowledge, we can now review the meta-object protocol of the language in the following section.

2.2.1.2 Meta-object protocol

An analysis of the CLOS meta-object protocol comprises two parts:

1. An analysis of the meta-level statics, i.e. the various types of meta-objects, each of which describes different base-level resources (e.g. classes, slots, methods, . . .).

2. An analysis of the meta-level dynamics, i.e. the actual *protocols* that describe how meta-objects interact with one another to achieve a certain result (e.g. the class initialization and finalization protocols).

Meta-level statics CLOS is composed of a set of basic building blocks, each of which is reflected upon in the meta-level class hierarchy. The main building blocks are: classes, slots, methods, generic functions and method combination. Each of these building blocks is reflected by a meta-object, instantiated from a meta-class. For example, the `class` meta-class is used to instantiate meta-objects that reflect upon base-level objects. The programmer can get hold of `class` meta-objects by using `find-class`. Meta-objects can be queried for their slots and their direct subclasses, amongst others. The code below illustrates this:

```
(setf personClass (find-class 'Person))
=> #<STANDARD-CLASS PERSON>
(class-slots personClass)
=> (#<STANDARD-EFFECTIVE-SLOT-DEFINITION NAME #x203A9C76>
    #<STANDARD-EFFECTIVE-SLOT-DEFINITION AGE #x203A9CBE>)
(class-direct-subclasses personClass)
=> (#<STANDARD-CLASS TEACHER>)
```

Meta-level dynamics Most operations in CLOS are the result of the execution of a well-defined protocol, i.e. a set of meta-level operations that are called in a specific order. A programmer who is aware of these protocols (the sum of which forms the meta-object protocol of CLOS) can use this knowledge to “hook in” at the right places to alter the default behaviour. For example, the slot reading protocol is defined by the `slot-value-using-class` generic function:

```
(SLOT-VALUE-USING-CLASS <CLASS-METAOBJECT> <OBJECT> <SLOT-NAME>)
1. Check for existence of slot
   (slot-exists-p <object> <slot-name>)
2. Check for slot being bound
   (slot-boundp-using-class <class-metaobject> <object> <slot-name>)
3. Retrieve the value
```

Hence, the programmer can alter the default slot reading behaviour by refining the `slot-value-using-class` method in a custom meta-class. To use this behaviour, it suffices to declare what meta-level class to use in association with a base-level class, as follows:

```
; a meta-level class
(defclass MyMetaClass (standard-class) ())
; a base-level class
(defclass Teacher (Person)
                  ((subjects :initform '()))
                  ; use MyMetaClass instead of standard-class
                  (:metaclass MyMetaClass))
```

Note that `standard-class` is a direct subclass of `class` that describes “regular” classes. Another direct subclass of `class` is `built-in-class`, which is used only as the meta-class of built-in classes such as `integer`, `symbol`, `string`, . . .

2.2.1.3 Example: adding support for persistent objects

The meta-object protocol of CLOS can be used to transparently implement persistent objects (i.e. objects whose state is stored in and retrieved from a database) [Pae93]. We will not go into the details here but only outline briefly the steps needed to extend CLOS with persistent objects. The general idea is to have a `persistent-metalevel-class` meta-class that base-level classes can use instead of the default `class` meta-class in order for their instances to be made persistent. A set of methods must be overridden in `persistent-metalevel-class` to handle the persistency feature of objects. For example, the `slot-value-using-class` method must be refined to lookup the value of a slot in the database if it is not cached in the object. Furthermore, the `initialize-instance` method of `class` must be refined to add some necessary "house keeping" slots to every persistent object, such as a `persistent?` slot that stores a boolean value indicating if the base-level object is a persistent object or not. This is achieved by transparently adding an object with the "house keeping" behaviour to the inheritance list of the persistent object[1]. The complete implementation of CLOS persistent objects can be found in literature [Pae93].

2.3 Mirror-based reflection

Although the reflection community advocates a strict separation between the base- and meta-level of a language architecture [Mae87], most mainstream object-oriented languages with reflective capabilities do not fully adhere to this principle. For example, in Java one might query an instance for its class and superclass by executing the following code [BU04]:

```
Class theCarsClass = aCar.getClass();
Class theCarsSuperclass = theCarsClass.getSuperclass();
```

While `getClass` and `getSuperclass` are conceptually meta-level operations, both methods are implemented at the base-level of the language. Indeed, `getClass` is implemented as a method of the `Object` root class, making every base-level class that inherits from `Object` exhibit meta-level behaviour and normal behaviour side by side. The same holds for the `Class` class, which contains constructors and static attributes alongside meta-level operations, such as the `getMethods` and `getSuperclass` methods. Hence, the base- and meta-level operations of languages like Java, CLOS, C#, Smalltalk, ... are inextricably entangled [BU04].

To solve this entanglement, mirror-based reflection [BU04] states that the base- and meta-level facilities of a reflective system must be separated from one another. This design principle is called the principle of *stratification*. Mirror-based reflec-

[1]This is, in fact, a form of mixin composition [BC90]. A mixin is a modular extension that may be "mixed" into several classes.

tion identifies three design principles for meta-level facilities in object-oriented languages [BU04]:

1. The principle of *stratification* described above. The advantage of stratification in a reflective language is that it makes it easy to eliminate reflection when it is not needed. One could also imagine a mechanism to dynamically add (remove) reflection support to (from) a running computation. Additionally, section 4.1.1 describes how the lack of stratification can cause name clashes between the base-level and the meta-level of a language.

2. The principle of *encapsulation*, which states that meta-level facilities must encapsulate their implementation. Hence, the clients of reflection should only have access to the public interface of meta-level objects; the particulars of the implementation of meta-level objects should be hidden and should only be available (if allowed) through a public interface.

3. The principle of *ontological correspondence*, which states that the ontology of meta-level facilities should correspond to the ontology of the language they manipulate. This comprises two aspects: a structural aspect and a temporal aspect. The former states that the structure of meta-level facilities should correspond to the structure of the language they manipulate. Hence, meta-object protocols should introduce meta-objects for every base-level entity, such as objects, but also methods, method bodies, statements, ... This aspect is called *structural correspondence*. The latter aspect, called *temporal correpondence*, states that meta-level modules should be separated based on the moment at which they operate. For example, code (compile-time data) and computations (run-time data) should be mirrored by separate modules of the reflective API.

To obey these design principles, special meta-objects called *mirrors* are introduced [BU04]. The API of a mirror-based reflective language would, for example, be designed as follows [BU04]:

```
class Object {
    // no reflective methods
    ...
}
class Class {
    // no reflective methods
    ...
}
interface Mirror {
    String name();
    ...
}
class Reflection {
    public static ObjectMirror reflect(Object o) { ... }
}
```

```
interface ObjectMirror extends Mirror {
    ClassMirror getClass();
    ...
}
interface ClassMirror extends Mirror {
    ClassMirror getSuperclass();
    ...
}
```

Hence, in a mirror-based system, the above example where an object is queried for its class and superclass would be rewritten as follows:

```
ObjectMirror theCarsMirror = Reflection.reflect(aCar);
ClassMirror theCarsClassMirror = theCarsMirror.getClass();
ClassMirror theCarsSuperclassMirror = theCarsClassMirror.getSuperclass();
```

The advantages of this mirror-based design over the previous design are three-fold [BU04]:

- Meta-level operations are pulled out in a separable subsystem. This corresponds to the principle of *stratification*. As stated above, this allows one to dynamically add (remove) reflection support to (from) a running computation. Another advantage of stratification between the base-level and the meta-level is that non-reflective applications written in reflective languages can be deployed on platforms without a reflective implementation, thereby reducing, amongst others, the memory footprint of the application.

- The interface to meta-level operations is divorced from a particular implementation. This embodies the principle of *encapsulation*. One possible advantage of encapsulation in a reflective architecture is that it makes it easier to write debugging tools that transparently debug local and remote objects. Indeed, if one programs the mirrors for local and remote objects to the same interface, the rest of the debugger's code can abstract from the different underlying implementations.

- The structure of the base-level is mirrored at the meta-level. This corresponds to the principle of *structural correspondence*. Imagine a reflective architecture that fully implements structural correspondence, by having meta-objects that reflect, amongst others, on statements and expressions. This would permit the clients of reflection to use a standardized representation of every entity in the language.

The differences between the previous design and the mirror-based design are illustrated in figure 2.1. Section 4.3 discusses the applicability of such a mirror-based design to our experimental distributed actor-based language *AmbientTalk* (which is discussed in chapter 3).

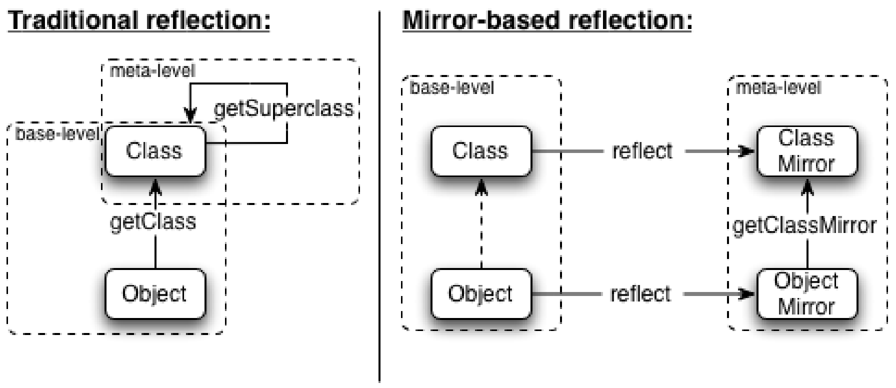


Figure 2.1: The differences between a traditional reflective design and a mirror-based design [BU04]

2.3.1 Case study: Self

The Self language [US87] uses a mirror-based design to support reflection. Although Self is a prototype-based language (i.e. classless: all objects are self-sufficient—see section 3.3.1), the ideas stated above are also applicable to Self (as a matter of fact, mirrors were first introduced in Self [BU04]). Indeed, the presence of classes at the base-level is not strictly necessary. As for CLOS, we begin with a brief overview of the base-level of the language.

2.3.1.1 Base-level

Self is an extremely simple yet very expressive language. In Self, everything is an object and every computation is triggered by sending a message to an object. Self distinguishes between three kinds of objects: "plain" objects, method objects and block objects [Meu04]. Plain objects can be created ex nihilo, by putting a number of slot names between vertical bars, possibly with an initial value (e.g. `| age <- 0. name = 'John' |`). The <- syntax creates a mutable slot (i.e. one whose value can be changed afterwards), while the = syntax creates an immutable (i.e. constant) slot. Method objects (further on referred to as "methods") can be created like so: `(| add: n = (value: value + n) |)`. This creates a method that takes one argument (`n`) and increments the `value` slot of the object on which it operates with the value of `n`. As a more concrete example, consider the following code which creates a stack object in Self [TK01]:

```
aStack <- (|
  stack = array clone.
  top <- 0.
  push: obj = (top: (top+1). stack at: top Put: obj).
  pop = (top: (top-1)).
  getTop = (stack at: top)
|)
```

The following section proceeds with an analysis of the reflective architecture of Self, which is based on mirrors.

2.3.1.2 Mirror-based reflective architecture

To obtain the mirror of an object in Self, one uses the `(reflect: base-obj)` method, where `base-obj` is the object to reflect upon. The returned mirror object understands, amongst others, the following messages:

```
reflectee    "Return the object being inspected"
size         "Return the number of slots in the reflectee"
names        "Return a vector of slot names"
at: slotName PutContents: mirror "Change the contents of SlotName
                                  to be the reflectee of mirror"
addSlot: slotMirror "Add a slot"
```

For example, one could reflect on the stack object defined above and call a number of reflective methods of its mirror:

```
(reflect: aStack) size
=> 5
aStack top
=> 1
(reflect: aStack) at: 'top' PutContents: (reflect: 0)
aStack top
=> 0
```

Hence, the Self reflective architecture supports introspection and structural intercession [BU04]. Furthermore, we observe that the principle of stratification is respected in Self: most[2] of a mirror's methods take mirror objects as arguments.

2.4 Reflection in actor-based concurrent languages

This chapter describes how object-oriented reflective architectures can be adapted to operate in a concurrent setting. To this end, it firstly explains the actor model, a model based on the notion of *active objects*. The actor model is used in e.g. ABCL/1, which we will discuss in section 2.4.2.1. ABCL/1 forms the base of the reflective concurrent object-oriented ABCL/R language described in section 2.4.2.2. Furthermore, the *AmbientTalk* language—our experimental platform for this dissertation—is based on an extended version of the actor model (see section 3.4).

[2]The first argument of the `at:PutContents:` method is a base-level string object, while its second argument is a (meta-level) mirror object. The reason for this is that Self's authors admit that it is not always clear whether a mirror method should accept a mirror or a base object as its argument [BU04].

2.4.1 The actor model

The actor model of computation [Agh86] defines a functional approach to concurrency. It is based on three main concepts: active objects (i.e. *actors*), asynchronous message passing between active objects and behaviour replacement [CM04]. Every actor is a self-contained unit of concurrency that has its own thread of control. Actors communicate with one another using asynchronous messages, meaning that nor the message send operation, nor the corresponding receive operation block. This contrasts with regular, synchronous messages in object-oriented languages, which block until the corresponding method returns. Instead, in the actor model, incoming messages are stored in an actor's message queue such that they can be processed later. Hence, every actor consists of the following entities [Agh90]:

- A *message queue* that acts as a buffer for incoming messages.
- A *behaviour* that denotes the set of methods and state variables of the actor.
- A *thread of control* that dequeues incoming messages from the message queue when approriate, and executes the behaviour corresponding to that message (generally a method specified in the actor's behaviour).

Additionally, the actor model is built upon three main primitives [Agh90]:

- The `create` primitive allows one to create an actor from a behaviour description and a set of parameters, that possibly includes existing actors.
- The `send` primitive allows one to send an asynchronous message to another actor. A call to `send` immediately returns and does not block until the result of the message is returned.
- The `become` primitive allows an actor to replace its own behaviour by a new behaviour. Hence, the way an actor responds to a message can change over time.

Whenever an actor receives a message, the corresponding method in its behaviour should specify a replacement behaviour to process the next message in the queue. Since no state is shared between behaviours, processing of message $n+1$ may start as soon as the replacement behaviour is specified by message n. Thus, it is possible for the processing of message $n+1$ to begin when the processing of message n is still running.

Furthermore, because message passing is purely asynchronous, actors are unable to explicitly return results to the sender of a message. To be able to return results, one can instead make use of *consumer actors*, passed as an extra argument to messages. These actors are meant to "consume" the result of the message. This, however, leads to event-driven applications whose code quickly becomes scattered and unreadable [CM04].

2.4.2 Case study: ABCL

This section analyzes the object-oriented concurrent language ABCL/1 [YBS86], followed by its reflective extension ABCL/R [WY88]. Both take root from the actor model described in the previous section.

2.4.2.1 Base-level: ABCL/1

ABCL/1 is a pragmatic approach to the functional actor model described in the previous section. It is based on *active objects* (or just *objects* in ABCL/1 terminology—not to be confounded with *passive* objects such as in CLOS and Self). Each active object in ABCL/1 consists of its own (autonomous) processing power and may have local persistent state. The behaviour of each object is described by a script. An object's script specifies what messages it accepts and what actions it performs in response to these messages [YBS86].

Differences with the functional actor model ABCL/1 active objects differ from functional actors in a number of ways:

- Each object can have its own persistent, updatable state.
- Objects process messages sequentially, in the order of arrival.
- Objects are always in one of the three following states: *active* (processing a message), *dormant* (not processing a message; empty message queue) or *waiting* (waiting for a certain message to arrive. For example, to process a `get` message when a buffer object is empty, the object must wait for an incoming `put` message).
- ABCL/1 supports multiple message passing types, including asynchronous but also synchronous message passing, as explained hereafter.

Message passing types ABCL/1 supports three types of message passing:

- *Past type message passing.* Messages are sent asynchronously, as in the functional actor model. Past type messages are denoted as follows: `[T <= M]`, where `T` is an object and `M` represents a message. An optional consumer object can also be specified: `[T <= M @ consumer]`.
- *Now type message passing.* Messages are sent synchronously, meaning that the sender blocks until the result of the message is returned. Now type messages are denoted as follows: `[T <== M]`.
- *Future type message passing.* Futures try to take advantage of the fact that the result of a message is not always directly needed, allowing for increased concurrency. Blocking futures (as well as their non-blocking equivalent) are described in more detail in section 3.5.1. Future type messages are sent

using the following syntax: `[T <= M $ x]`, where x is a “placeholder” for the result of the message.

Ordinary mode and express mode messages In addition to ordinary mode messages as described above, one can also send express mode messages to an actor. To support this, every actor has an additional *express message queue* besides its ordinary message queue for ordinary mode messages. Just like ordinary mode messages, messages in the express message queue are processed sequentially. If an express mode message arrives while an object is processing an ordinary mode message, the latter computation is suspended and the express mode message is immediately processed. Afterwards, the processing of the ordinary mode message is resumed (unless explicitly aborted by the express mode message). All message passing types support the express mode by adding a second < to the message sending statement. For example, to send an express mode now type message, one writes: `[T <<== M]`.

Example The following example code illustrates how a simple alarm clock object can be implemented in ABCL/1 [YBS86].

```
[object Ticker
  ;; persistent state variables of the object
  (state [time := 0] [alarm-clocks-list := nil])
  ;; behaviour of the object
  (script
    (=> [:start]
      (while t do
        (if alarm-clocks-list
         then [alarm-clocks-list <= [:tick time]])
        [time := (1+ time)]))

    ;; :add can only be sent in express mode
    (=>> [:add AlarmClock]
      [alarm-clocks-list := (cons AlarmClock alarm-clocks-list)])

    (=>> [:stop] (non-resume)))]

[object CreateAlarmClock
  (script
    (=> [:new Person-to-wake]
      (temporary
        ;; create alarm clock object
        [AlarmClock := [object
                         (state [time-to-ring := nil])
                         (script
                           (=> [:tick Time]
                             (if (= Time time-to-ring)
                              then [Person-to-wake <<= [:time-is-up]]))
                           (=> [:wake-me-at T]
                             [time-to-ring := T]))]]
        ;; register alarm clock at ticker (express mode message)
        [Ticker <<= [:add AlarmClock]]
        ;; return alarm clock object
        !AlarmClock)))]
```

2.4.2.2 Meta-level: ABCL/R

ABCL/R [WY88] adds a powerful reflective architecture to ABCL/1. To support reflection in ABCL/R, the structural and behavioural aspects of each object x are reflected upon by a meta-object $\uparrow x$. The meta-object of x contains information that refies its *denotation*'s (i.e. its base-object's) message queue, evaluation thread and its state and behaviour. The following points are worth noting [WY88]:

- x and $\uparrow x$ are causally connected. Thus, the data stored in $\uparrow x$ represent the current status of x at any time.
- Since $\uparrow x$ is an object, $\uparrow\uparrow x$ also exists to reflect upon $\uparrow x$. Hence, there is an infinite tower of meta-objects $\uparrow x, \uparrow\uparrow x, \uparrow\uparrow\uparrow x, ...$ for each base-level object x. In the implementation, this infinite tower is "short-circuited" by creating meta-objects only when they are accessed (i.e. lazy creation). This solution is often adopted by object-oriented reflective languages to support an infinite tower of meta-objects at the implementation level.
- There is a one to one correspondence between a meta-object $\uparrow x$ and its denotation x.

Meta-circularity of objects Like many reflective language architectures, ABCL/R is based on a meta-circular design [Mae87]. Indeed, the meta-object $\uparrow x$ is used as the actual implementation of x. The following excerpt from the definition of a default meta-object illustrates this [WY88]:

```
[object ;; a meta-object
  (state [queue := a message queue]
         [state := a state object]
         [scriptset := a list of scripts]
         [evaluator := an evaluator object]
         [mode := either :dormant or :active])
  (script
    (=> [:message Message Reply-Dest Sender] ;; message receiving
        [:queue := (enqueue queue [Message Reply-Dest Sender])]
        (if (eq mode :dormant)
         then [mode := ':active]
              [Me <= :begin])) ;; Me points to this object
    (=> :begin ;; message processing
        (temporary mrs scr newenv [object := Me])
        [mrs := (first queue)]
        [queue := (dequeue queue)]
        [scr := (find-script (first mrs) scriptset)]
        (if scr
        ;; accept message
        then [newenv := [env-gen <== [:new (script-alist mrs scr)
                                             state]]]
             ;; process the message using the corresponding method
             ;; from the denotations script, and ignore the value
             ;; of the evaluation
             [evaluator <= [:do-prg (scr$body scr) newenv [den Me]]
                       @ [cont ignore
                           [object <= :end]]]
        ;; cannot accept message (no behaviour found)
        else (warn "Cannot handle message" (first mrs))
             [Me <= :end]))
   (=> :end ;; termination of execution
       (if (not (empty? queue))
        then [Me <= :begin]
        else [mode := :dormant]))
   ...)]
```

As the above code indicates, the `queue`, `evaluator`, `mode`, `state` and `scriptset` variables of a meta-object reify (and are used as the implementation of) its denotation's message queue, evaluator, mode, persistent state and behaviour, respectively. Similarly, the `:message`, `:begin`, `:end`, ... methods are used for implementing message arrival, message processing, termination of message processing, ... For example, the body of the `:begin` method reveals that messages are processed sequentially, by dequeueing the first message and looking up the corresponding method in the denotation's scriptset. If the method is found, its body is executed in a newly created environment by sending a past type message to the `evaluator` object. After the execution of the body, the meta-object is sent an `end:` message, to indicate that it can process the next message (if any). Because a past type message is sent to the evaluator, $\uparrow x$ immediately returns to dormant mode while x stays in active mode until the end of the execution of the method's body. Hence, $\uparrow x$ can enqueue new incoming messages for x while x is concurrently performing a computation. This is called the *inherent concurrency* of x [WY88], and corresponds to the semantics of message receiving and processing as described in section 2.4.2.1.

Linking the base-level to the meta-level One can query an object `x` for its meta-object by executing `[meta x]`. Similarly, one can query a meta-object `m` for its denotation by executing `[den m]`. This allows for structural intercession. Indeed, x can modify itself through its meta-object $\uparrow x$ (since the base- and meta-level are causally connected). Other objects that contain a reference to x can also obtain its meta-object $\uparrow x$ and use it to modify x (for example to add/modify methods to/of x, as in the example below).

To allow for behavioural intercession, one can create an object and specify a custom meta-object as its implementation, like so:

```
[object
  (meta custom-meta-object)
  (script ...)]
```

Example Suppose that a pool of objects is used to execute jobs (for the sake of our example, each job is an expensive computation). Suppose also that there is *manager* object M which deals out jobs uniformly to every *worker* object $W_1, \ldots, W_n$, by sending a `[:job <job-type> :param <parameter>]` message. M can now monitor every worker object and at any time change the implementation of the job-processing method of W_i with a more optimized method, using the following code:

```
[[meta Wi] <== [:add-script
      '(=> [:job 1 :param <parameter-var>]
           <optimized-method-body>)]]
```

Note that the method redefinition of W_i can be done transparently while W_i is executing its jobs, because of the inherent concurrency of W_i, as described above. This inherent concurrency does, however, introduce possible race conditions, for example if a method m is removed from the behaviour of x through $\uparrow x$ while x is still executing m.

2.5 Conclusion

Throughout this chapter, we have studied computational reflection, a powerful mechanism that allows the programmer to customize a language to his needs without necessarily recompiling the code of the language interpreter. Reflection is achieved by reifying language constructs and behaviour, so that the programmer can hook in at certain points to add specific behaviour to the interpreter. We have started our overview with an analysis of meta-object protocols, which are the *de facto* way of supporting behavioural intercession in an object-oriented language (as an example, we have seen that slot access in CLOS can be overridden with custom behaviour using the language's meta-object protocol). Subsequently, we have looked at mirrors, which define a set of sound design principles for maintainable object-oriented meta-level architectures. Finally, we have studied how structural and behavioural intercession can be achieved in a concurrent object-oriented language based on the functional actor model. Every reflective design principle was extensively illustrated using an existing reflective language architecture as a case study, namely CLOS, Self and ABCL.

The next chapter introduces a new programming paradigm called *ambient-oriented programming*, which is especially tailored to meet the stringent requirements imposed by the hardware characteristics of mobile networks. It also introduces AmbientTalk—the experimental language used throughout this dissertation. AmbientTalk is based on the ambient-oriented actor model, an extension of the functional actor model already outlined in this chapter.

All the design principles described in this chapter will prove to be important when designing an enhanced meta-object protocol for AmbientTalk in chapter 5.

3 Ambient-Oriented Programming

This chapter describes an emerging new programming paradigm called "Ambient-Oriented Programming", which aims to simplify the development of applications designed to run on mobile networks.

It starts with a definition of mobile networks and an overview of the main properties that characterize them. These properties induce a set of requirements for the ambient-oriented paradigm. Section 3.4 introduces AmbientTalk, an experimental programming language based on the ambient-oriented paradigm. Recall from chapter 1 that the goal of this dissertation is to design a mirror-based meta-object protocol (see section 2.3) for an ambient-oriented programming language, namely AmbientTalk.

This chapter ends with a review of AmbientTalk's existing meta-object protocol—which will be evaluated in detail in chapter 4—and various language extensions, implemented using the meta-object protocol.

3.1 Introduction

The last years have seen a rapid emergence of wireless computational devices and the mobile networks they imply. Due to their differences compared to stationary networks of wired devices, existing programming paradigms are showing their limitations when used for programming such mobile devices. As a result, the need for a new programming paradigm, designed from the ground up with the main characteristics of mobile networks in mind, is becoming evident. This paradigm has been named "Ambient-Oriented Programming" [DVM$^+$06]. The next section defines *mobile networks*.

3.2 Characteristics of mobile networks

We define mobile networks as the networks that surround wireless devices and that enable them to interact with each other. The best example of mobile networks to date is Wi-Fi (IEEE 802.11), a standard for wireless networks that allows laptops, PDAs, printers, ... to communicate with each other.

Mobile networks have a number of features that distinguish them from stationary networks (e.g. Ethernet, Token ring, ...). For example, due to the limited transmission range of mobile devices, connections between two hosts in a mobile network are very volatile. Furthermore, mobile networks are said to be open, because devices can appear or disappear at any time without prior warning. The main differences between mobile and stationary networks are summarized below [DVM$^+$06]:

3.2.1 Connection persistence

Due to the very nature of wireless networks, connections between devices in a mobile network are much more volatile than in a stationary network. Whereas disconnections are rather rare in stationary networks, they are the rule rather than the exception in mobile networks: every time a wireless device moves out of the limited communication range of a transmitter, a disconnection occurs. However, many applications expect automatic reconnections after disconnections to be handled transparently.

3.2.2 Resource availability

Due to the limited transmission range of mobile devices, resources that are acquired can go out of range at any moment without any warning whatsoever. This is in strong contrast with stationary networks in which references to remote resources are obtained based on the explicit knowledge of the availability and location (cfr. URLs) of the resource. Hence, mobile devices often rely on a service discovery protocol to discover other resources on the network. Examples of existing service discovery protocols include *Jini* [Arn99] for Java objects and *UDDI* [Bel] for web services.

3.2.3 Autonomy

The predominant paradigm for writing distributed applications running on stationary networks is based on a client-server protocol. In mobile networks, however, devices cannot always rely on a central server with a high availability. For this reason, every mobile device should be able to act autonomously. Hence, ambient-oriented applications should be based on a peer-to-peer model, in which every device acts as a combination of a client and a server.

3.2.4 Concurrency

In distributed applications based on a client-server model, clients often use *remote procedure calls* (RPC) or *remote method invocation* (RMI) to interact with a server. Both mechanisms use synchronous communication. This means that every time the client calls a procedure (resp. invokes a method) on the server, the client blocks until the result of the procedure (resp. the method) is returned. However, blocking is undesirable in the context of mobile networks, as both the client and the server can go out of each other's range anytime.

This leads to the observation that concurrency is a more natural phenomenon in mobile networks, since the autonomy of a device is undermined every time it blocks while waiting for a result from a remote device.

3.3 Language criteria for an ambient-oriented programming language

Based on the main features of mobile networks cited above, we can define which language criteria are suitable for an ambient-oriented programming language. For example, due to the high volatility of connections, we cannot handle disconnections using traditional exceptions because this would clutter the code with numerous try-catch clauses. This cluttering would result in unreadable, difficultly maintainable and error-prone program code.

More generally, the ambient-oriented programming paradigm embodies four important language criteria [DVM$^+$06]:

3.3.1 Classless object models

In class-based programming languages such as Java and C++, every object is intrinsically linked to its class and cannot exist without it; the class defines the behaviour of the object and is shared by all of its instances. Although class-based languages may perform well in non-distributed applications, problems arise in a distributed context. Indeed, suppose that n devices host objects that are instances of a particular class C, then each host will need a local copy of the class. Consistency issues will occur as soon as one device, say n_i changes its local definition of C. Instances of the class C which are sent back and forth between device n_i and other devices would exhibit different behaviour depending on the device on which they reside, because objects are not self-contained, but instead depend on a class.

An alternative to class-based models are prototype-based models [DMC92], which shun the concept of a class. In prototype-based languages like Self [US87] and Agora [CHDS94], objects can be made self-sufficient—they can encapsulate all of their state and behaviour. This implies that, in prototype-based languages, there are three modi operandi to instantiate new objects:

1. By *cloning* an existing object (the existing object is used as a prototype). Cloning offers two alternatives: shallow cloning or deep cloning. However, deep cloning is often ruled out because it is more time consuming and offers little advantages over shallow cloning [DMC92].

2. *Ex nihilo*, i.e. by assembling a number of fields and methods to create a new object.

3. By *extending* an existing object p. In this scenario, a new object is created with a delegation link to its *parent* object p. This is comparable to class-based inheritance: if the object does not contain a field or a method x, the delegation links are followed until x is found in a parent. When following the delegation chain, the pseudo-variable `this` always points to the initial receiver of the message (i.e. late binding) [DMC92].

Because of the self-containment principle, classless models are more appropriate to the ambient-oriented paradigm. Indeed, the consistency issue described above does not arise in classless models, because no object depends on a class. Hence, two hosts can independently "upgrade" objects cloned from the same object, without risking that one of these objects behaves differently depending on where it is hosted.

3.3.2 Non-blocking communication

As we have seen in section 3.2.4, synchronous (blocking) communication between devices of a mobile network is undesirable because it severely undermines the autonomy of the blocked device, which, in the event of a network failure, could be waiting for an answer forever (or at least until a specified timeout). Additionaly, blocking communication is a known source of (distributed) deadlocks [VA98].

For these reasons, we conclude that ambient-oriented programming should use non-blocking communication primitives, in which neither the sender nor the receiver block until the result is returned (resp. until a message is received). Non-blocking communication leads to event-driven applications, responsive to incoming events generated by spontaneously interacting autonomous devices [DVM+06].

3.3.3 Reified communication traces

With non-blocking communication in place and the absence of synchronous communication primitives, devices lack an implicit way to synchronize with one another. However, synchronization is an essential component in distributed applications because it can, amongst others, prevent communicating devices from ending up in inconsistent states.

When the communication traces (e.g. the list of sent and received messages) are accessible to the programmer (i.e. reified in the language), they can be used to revert to an older, consistent state, or to a new state agreed upon by the communicating devices.

Reifying the communication traces and primitives is also important in the context of allowing the programmer to alter the default message sending or delivery policy. This allows the programmer to adapt the message delivery policy to his needs—and to the reality of the network environment. For example, in a real-time application the programmer might want to use “send once” semantics without any delivery guarantees instead of trying to send a message repeatedly until it reaches its destination.

3.3.4 Ambient acquaintance management

The volatility of connections and the unpredictable availability of resources makes it impractical to rely on a third party (i.e. a server) to get explicit references to other resources. Thus, devices should be able to discover other devices based on a description (e.g. an interface that the device sought after must implement), rather than on a fixed URL. One way to achieve this is to broadcast periodically the services that a device offers, such that other devices can get acquainted if they need the services offered by the device.

Hence, ambient devices should form a peer-to-peer network in which devices can spontaneously get acquainted with previously unknown devices, based on an intensional description of a set of required services, rather than on a fixed URL. Such a service discovery protocol, along with a mechanism to detect and handle the loss of acquaintance (e.g. whenever the provider and the consumer of a particular service move out of each other’s range), should be part of every language that implements the ambient-oriented programming paradigm [DVM+06].

3.4 AmbientTalk: an ambient-oriented programming laboratory

AmbientTalk is a programming language developed at the *Programming Technology Laboratory* of the *Vrije Universiteit Brussel* which adheres to all of the above language criteria. It serves as a "language laboratory" to experiment with different language constructs, such as futures, which are described in more detail in section 3.5. Thanks to its simple yet expressive meta-object protocol, most of these language constructs can be implemented in AmbientTalk without adapting the code of the language interpreter. We first introduce the language and its key concepts.

3.4.1 Pic% background

The AmbientTalk language is based on Pic%, which is itself an extension to Pico [D'Hb]. Pico is a simple procedural language that was designed to teach advanced computer science concepts to college students [D'Ha]. Pic% grew out of Pico, with the main difference being the addition of first-class dictionaries, which can be used to support prototype-based object-oriented programming. Pico's syntax and design rationale are described below, followed by a section which illustrates the object-oriented extensions offered by Pic%.

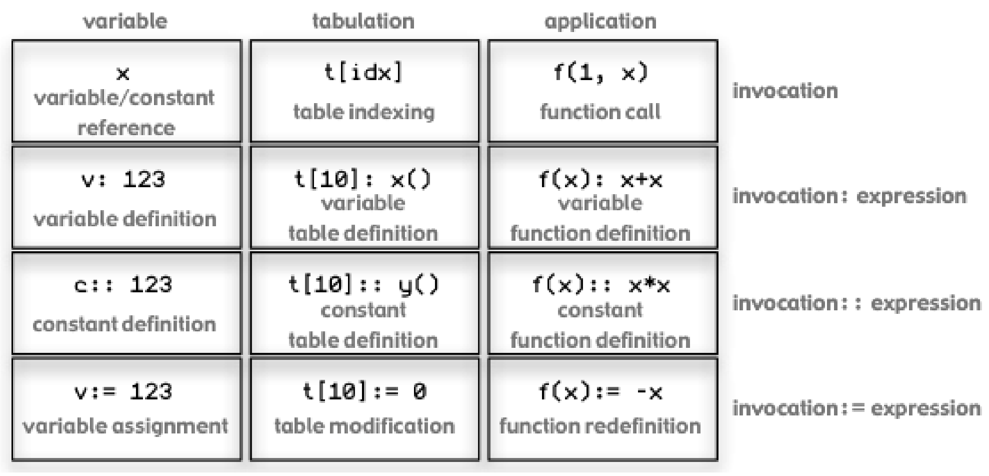
3.4.1.1 Pico

Below are some syntactical and semantical aspects of Pico, as explained in greater detail in [MDD04].

Table-based Whereas Scheme [ASS85] uses lists to store most of its data structures (including program code itself), Pico uses tables (i.e. arrays) consistently across the language: tables are used to store programs, tables are the basic data structures of the language, variable size argument lists are implemented by table parameter passing, the memory model and garbage collector are optimized to handle variable sized tables, ... [D'Hb]. Also noteworthy is the fact that Pico tables are indexed in the range $[1, ..., size(table)]$ instead of the usual $[0, ..., size(table)[$ range.

A Straightforward syntax Pico features a rich but nonetheless simple syntax. A fundamental notion of Pico is the *invocation*, which is either a reference, a tabulation or an application. An invocation is used in four modes: access, variable definition, constant definition and assignment. Figure 3.1 depicts this structure and illustrates how twelve different Pico program expression types are constructed by combining the three invocation types into the four modes.

No special forms by means of call-by-function Unlike Scheme, Pico does not require special forms in order to support block structures, recursion, conditionals, ... Instead, it features a parameter binding mechanism called *call-by-function*, which offers the programmer the option of specifying a formal parameter as an invocation. Consider the following illustrative example of a Scheme-like `map` function:



variable	tabulation	application	
x variable/constant reference	t[idx] table indexing	f(1, x) function call	invocation
v: 123 variable definition	t[10]: x() variable table definition	f(x): x+x variable function definition	invocation: expression
c:: 123 constant definition	t[10]:: y() constant table definition	f(x):: x*x constant function definition	invocation:: expression
v:= 123 variable assignment	t[10]:= 0 table modification	f(x):= -x function redefinition	invocation:= expression

Figure 3.1: Pico 3*x*4 syntax grid [MDD04]

```
map(f(val), tab):: {
  idx: 0;
  res[size(tab)]:: f(tab[idx:=idx+1])
}
```

A call to `map(val*val, [1, 2, 5])` evaluates to `[1, 4, 25]`. In this particular example, the call-by-value parameter `tab` is bound to the result of evaluating `[1, 2, 5]` in the calling scope, while the call-by-function parameter `f(val)` is bound to the expression `val*val`. Internally, during every application of `map`, a local variable f is bound to a closure consisting of the parameter list `(val)`, the body `val*val` and the calling evironment of `map` [MDD04].

Variable arity functions Pico features variable arity functions by means of the `@` construct. The following example illustrates its usage:

```
print@elements:: {
  for(i:1, i <= size(elements), i:= i+1,
      display(elements[i]))
}
```

This allows the programmer to call `print` with any number of arguments. Hence, `print(1)` and `print(1, 2, 3)` are both correct function calls. In both examples, the parameter `elements` is bound to a table containing one and three values, respectively.

3.4.1.2 Pic%

Pic% extends the syntax and the semantics of Pico in various ways, as explained below.

First-class environments for abstraction Pic% features first-class environments: the current environment can be captured at any moment using the `capture()` native. Internally, an environment consists of two linked lists of `(name, value)` pairs—one for the constants and one for the variables—as well as a link to a parent environment. This structure is depicted in figure 3.2.

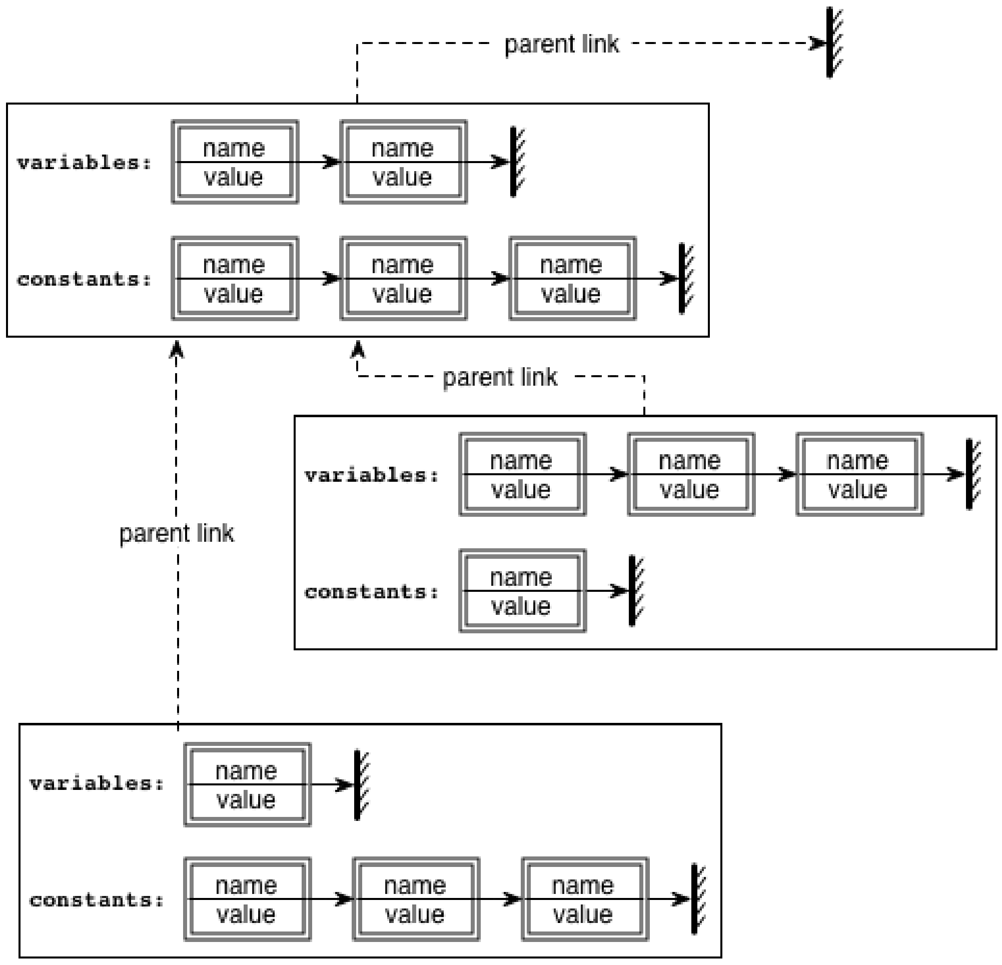


Figure 3.2: Structure of Pic% environments

Because of these properties, environments can be used as objects to enable object-oriented programming in Pic%. In some aspects, this is similar to the way objects are “simulated“ in Scheme [ASS85]. As an example, consider the following definition of a stack in Pic% [MDD04]:

```
Stack(n):
  { T[n]: void;
    t: 0;
    empty():: t = 0;
    full():: t = n;
    push(x):: { T[t:= t+1]:= x; void };
    pop():: { x: T[t]; t:= t-1; x };
    capture() }
```

Extended syntax and native library Pic% extends Pico with new syntax: the `obj.msg()` syntax is used to send messages to objects, while the `.msg()` syntax is used for super sends. All constants are public and all variables are private. Hence, only constants can be accessed using the dot notation. Additionally, the `this()` native always returns the current object and the `super()` native returns the parent of the current object.

Prototype-based extensions Recall from section 3.3.1 that there are three mechanisms for creating new objects in the prototype-based paradigm. Pic% supports all three mechanisms:

1. *By cloning an existing object.* The `clone(obj)` native returns a clone of `obj`. Cloning implies deep copying all the variables and shallow copying all the constants of a given object. Alternatively, Pic% supports *cloning methods*, i.e. methods whose body is always executed in the context of a clone of the receiver. Every method that starts with `cloning.` is a cloning method.

2. *Ex nihilo.* The `object(...)` native allows the programmer to create new objects ex nihilo. It creates an object by executing its argument expression, typically a block of code containing a number of slot declarations.

3. *By extending an existing object* The `obj.extend(...)` construct creates a new object from a number of slot declarations, with a delegation link to `obj`.

The following example illustrates various object creation mechanisms, as well as method overriding in Pic%:

```
makeCollection()::object({
  contents: [];
  setContents(elements):: { contents := elements };
  contains(el):: {
    result: false;
    for(i: 1, i <= size(contents), i:= i + 1,
        result:= or(result, contents[i] = el));
    result }
})

makeStringCollection(elements):: {
  c: makeCollection().extend({
    cmp(s1, s2):: ...; ` very expensive operation `
    contains(substr)::
      or(.contains(substr),
        { result: false;
          for(i: 1, i <= size(contents), i:= i + 1,
              result:= or(result,
                             cmp(contents[i], substr)));
          result });
    cloning.new(elements):: { contents:= elements }
  });
  c.setContents(elements)
}
```

We note that, being a prototype-based language because of its Pic% roots, AmbientTalk already satisfies the first criterion for an ambient-oriented programming language (see section 3.3.1).

With the knowledge of Pic% in mind, we are primed to continue with an analysis of the ambient-oriented features of AmbientTalk.

3.4.2 AmbientTalk's object model

To support the ambient-oriented programming paradigm, AmbientTalk uses the concept of *active objects* (or *actors*—we will use both terms interchangeably throughout the remainder of this document). Like the ABCL/1 actor model described in section 2.4.2.1, AmbientTalk's actor model is based on the functional actor model presented in section 2.4.1.

Briefly put, each actor is an "active" object, consisting of its own thread of execution, updatable state and methods and a message queue. Actors communicate with each other through asynchronous messages which are added to the actor's message queue. Messages in an actor's incoming queue are then processed sequentially to avoid race conditions. The following sections focus in more detail on the different parts of the AmbientTalk actor model.

3.4.2.1 State and behaviour

Actors are created using the `actor(bhv)` native, where `bhv` is a passive object which dictates the state and behaviour of the newly created actor. To eliminate possible race conditions that could occur when sharing the same passive object between different actors, AmbientTalk uses a deep copy of `bhv` as the behaviour of the actor. This way, no behaviour is ever shared by the new actor and its creator.

The behaviour of an actor can also be changed at any time by sending the actor a `become(newBhv)` message. Again, the argument of the `become` message (the passive object describing the new behaviour) is deep copied to prevent race conditions.

3.4.2.2 Message processing and sending

Actors communicate with each other using asynchronous messaging: neither does the sender block until the answer to a message is returned, nor does the receiver block for incoming messages. This satisfies the *non-blocking communication* the criterion of ambient-oriented programming languages (see section 3.3.2). Again, to prevent race conditions, all message arguments with the exception of actors[1] are deep copied, such that a passive object is never shared by two active objects.

Thus, the functional actor model satisfies the first two criteria of the ambient-oriented paradigm so far (*classless object model* and *non-blocking communication*). However, the last two criteria—*reified communication traces* and *ambient acquaintance management*—are unsatisfiable in the "regular" actor model [MTM$^+$05]. Therefore, AmbientTalk is based on an extension of Hewitt and Agha's actor model [Agh86], called the ambient actor model [DB04]. The ambient actor model replaces the single message queue of the functional actor model with a number of first-class mailboxes, as detailed below.

To support the *reified communication traces* criterion (see section 3.3.3), AmbientTalk replaces the single message queue of every actor with four first-class mailboxes (i.e. mailboxes that are explicitly accessible to the programmer): an `inbox`, an `outbox`, a `rcvbox` and a `sentbox`. Whenever an actor `a` sends a message `m` to another actor `b` (using the `b#m(arg, ...)` syntax), the message `m` is stored indefinitely in the `outbox` of `a` until it is successfully transmitted to `b`. At that moment, the message `m` is moved from `a`'s `outbox` to `a`'s `sentbox`. On `b`'s side, the incoming message `m` is kept in the `inbox` until it is processed. After it has been processed, `m` is moved from `b`'s `inbox` to `b`'s `rcvbox`.

Together, these four mailboxes fully reify the communication traces between actors (see section 3.3.3), as required by the ambient-oriented programming paradigm.

3.4.2.3 Service discovery

Apart from the four mailboxes used as message queues, every actor contains four additional mailboxes that are used for service discovery. Every actor can add descriptive strings to its `provided` mailbox to announce one or more provided services. Similarly, an actor can put a series of descriptive strings in its `required` mailbox to announce that it is seeking actors providing those services for collaboration. Whenever two actors come into each other's range, the descriptive strings of both actors are matched against each other. If a match occurs, the `joinBox` mailbox of the actor requiring the service *s* is updated with a resolution object containing a reference to the actor providing service *s* and the matched descriptive string. Every time two such actors go out of range, the resolution object is moved from the

[1]Actors process messages sequentially, thus eliminating the possibility of race conditions.

joinBox mailbox to the disjoinBox mailbox. This is the core of AmbientTalk's acquaintance management mechanism (see section 3.3.4).

3.4.2.4 Mailbox observers

The programmer can register an observer for each native message mailbox (inbox, outbox, sentbox and rcvbox) by defining in the actor's behaviour the in(msg), out(msg), sent(msg) and rcv(msg) methods, respectively. Mailboxes observers are called every time a message is added to a mailbox, with the message as their unique argument.

As a simple example of the use of mailbox observers, consider a *router* actor that forwards every incoming message transparently to another actor. The router actor also logs every forwarded message. The code is given below:

```
router: actor(object({
    target: void;
    log: vector.new();

    setTarget(act):: { target := act };

    in(msg):: {
      if(not(is_void(target)), {
          ‘ log the message and the current time ‘
          log.add([time(), msg]);
          ‘ forward the message transparently ‘
          inbox.delete(msg);
          msg.setTarget(target);
          outbox.add(msg)
      })
    }
}))
```

3.4.2.5 Meta-object protocol

AmbientTalk supports structural and behavioural intercession (see section 2.1) by reifying parts of an actor's internal structures and behaviour, which are implemented as regular Pic% passive objects and methods. While this may have negative effects on the execution speed of AmbientTalk programs, it has the advantage of opening up the implementation of e.g. the message creation and transmission mechanisms to the programmer. For example, every asynchronous message is created as the result of a call to the createMessage method of the sender's behaviour. This yields a passive object containing the source (the actor that sent the message), target (the receiver of the message), name (the name of the message) and argList (a table containing all the arguments of the message) slots and a process method. The default implementation of createMessage is given below:

```
createMessage :: { ‘ MOP method to create new messages ‘
```

```
  message : object({
    source : void;   target  : void;
    name   : void;   argList : void;

    cloning.new(aSource, aTarget, aName, anArgList)::{
      source:=aSource;   target:=aTarget;
      name:=aName;       argList:=anArgList
    };

    getSource()::source;   setSource(aSource)::source:=aSource;
    getTarget()::target;   setTarget(aTarget)::target:=aTarget;
    getName()::name;       setName(aName)::name:=aName;
    getArgs()::argList;    setArgs(anArgList)::argList:=anArgList;

    process(behaviour)::behaviour.execute(this());
  });

  message.new
}
```

Similarly, an actor sends an asynchronous message by calling the `send` method defined in its behaviour. The default implementation of `send`, as included in the root object to which all objects delegate (either directly or indirectly), is given below:

```
send(message):: {
  outbox.add(message);
  void
}
```

As part of the meta-object *protocol*, every expression `act#msg(arg, ...)` is translated internally to `send(createMessage(thisActor(), act, "msg", [arg, ...]))`.

Processing messages that are removed from the inbox is for its part handled by the `process` method, implemented by default as:

```
process(message):: {
  message.process(this());
  rcvbox.add(message)
}
```

The bodies of these methods clearly map to the semantics of message sending and processing as explained in section 3.4.2.2.

The following example illustrates the usage of the MOP, by demonstrating how the `send` method can be refined for debugging purposes [DVM+06]:

```
send(msg):: {
  display("Sending '", msg.getName(), "'", eoln);
  .send(msg)
}
```

Please note that the built-in mailboxes and their observers are also part of AmbientTalk's MOP, since they partially reify the state of the interpreter. Hence, each stage of the message creation, sending and processing interplay is reified in the MOP [DVM+06], allowing the programmer to intervene wherever required. As we will see in the following section, the MOP is used extensively to reflectively implement custom language extensions, such as futures and due-blocks.

3.5 AmbientTalk language extensions

As explained above, AmbientTalk serves primarily as a laboratory for experimenting with ambient-oriented language constructs. This is usually accomplished by using and overriding the MOP methods described in the previous section. A mixin-based technique is used to implement the language constructs in a modular way: an AmbientTalk language construct and its supporting MOP methods are grouped in a *language extension* (or *language mixin*), i.e. a method that extends its argument (a passive object describing an actor's behaviour) with new meta-level behaviour (e.g. by overriding `send`, `createMessage`, `process` and/or installing mailbox observers). The language mixins are to be applied to the passive object describing the actor's behaviour before it is created, such that the newly created actor can exhibit the required additional behaviour [DVM+06]. As an example, two AmbientTalk language extensions are analyzed below.

3.5.1 Non-blocking futures

Futures were developed to reconcile asynchronous message sends with classic synchronous method call return values, and to avoid the drawbacks of event-driven programming[2] (i.e. cluttered code, race conditions, ...). The idea is that an asynchronous message send results in a placeholder object (i.e. a "future") which will eventually be replaced by the actual result of the asynchronous message send (i.e. the "resolution"). Because some operations such as assignment and parameter passing operate only on the variables and not on the values themselves, parallelism between caller and callee can increase significantly. Operations which do need the result (e.g. addition) are blocked until the future is resolved[3]. For example, consider the following example code in MultiLisp [RHH85] (the `future` construct returns a placeholder that is replaced by the resolution as soon as it is computed):

```
(cons (future (fib (- n 1))) (future (fib (- n 2))))
(+ (future (fib (- n 1))) (future (fib (- n 2))))
```

Clearly, the first expression (which creates a pair of values) has more potential for parallelism than the second expression, because the + operator will immediately need to examine its argument values (so it can add them) whereas placeholders are

[2]Recall from section 3.3.2 that non-blocking communication primitives give rise to event-driven applications.

[3]In the optimal case, the future has already been resolved such that no blocking occurs at all.

sufficient for the `cons` operator [RHH85].

AmbientTalk's futures, however, differ from "traditional" (i.e. blocking) futures, and are instead based on the concept of non-blocking futures introduced in E [MTS05]. The reason for this is that blocking futures can undermine the autonomy of ambient devices, should a computation be waiting for the resolution in order to be able to continue.

Hence, AmbientTalk features non-blocking futures which transparently forward messages to their resolution, and a `when(aFuture, aClosure)` construct that executes a given closure upon resolving the future. An example of AmbientTalk's implementation of non-blocking futures is given below.

3.5.1.1 Example

To enable the use of futures, the programmer extends the desired behaviour (`facBhv`) with the "future-enabling" behaviour. In the following example, the asynchronous message send `facActor#fac(i-1)` returns a non-blocking future (instead of `void`). The `when` construct takes two parameters: the first is a non-blocking future, and the second is a block of code that will be executed at the moment the future is resolved. This block of code may be parametrized with a `content` parameter, which will be bound to the value of the resolution at the moment of execution (i.e. when the future is resolved).

```
{
  facActor:: actor(object(
      fac(n):: if(n = 0, 1, n * fac(n-1))
  ));

  facBhv:: object({
      printFac(i):: {
        when(facActor#fac(i-1),
             display("fac(", i, ") = ", content * i, eoln))
      }
  });

  act: actor(extendWithFuturesBehaviour(facBhv));
  act#printFac(42)
}
```

3.5.1.2 Reflective implementation

The AmbientTalk code implementing non-blocking futures consists of three parts: the behaviour of the future actor, the behaviour of a listener actor which gets notified when the future is resolved, and the behaviour to add to "regular" actors to make them capable of handling futures (a.o. introducing the `when` language construct).

Future-enabling behaviour The `extendWithFuturesBehaviour(bhv)` method is used to extend the behaviour of a regular actor so that it can handle futures correctly. Firstly, `createMessage` is refined in order to create extended messages which contain a reference to a future and additional behaviour to resolve this future when the result of the computation induced by the asynchronous message is available. Secondly, the `send` method is refined such that it returns the future instead of `void`. Finally, the `when` method is added to an actor's behaviour to allow the programmer to specify a block of code to execute as soon as a future is resolved. Implementation-wise, this is achieved by subscribing a listener actor to the future. The behaviour of the listener actor is explained below.

```
extendWithFuturesBehaviour(bhv):: bhv.extend({
    whenBlocks: vector.new();
    newId     : 1;

    invokeWhen(anId, content)::{
      whenBlocks.get(anId)(content)
    };

    when(aFuture, code(content))::{
      whenBlocks.add(code);
      aFuture#subscribe(actor(futureListener.new(newId, thisActor())));
      newId:=newId+1;
      void
    };

    createMessage(aSource, aTarget, aName, anArglist)::{
      ‘ create a regular message ‘
      msg: .createMessage(aSource, aTarget, aName, anArglist);
      ‘ ... and extend it with future behaviour ‘
      msg.extend({
          future: void;

          getFuture()::future;
          setFuture(aFuture)::future:=aFuture;

          process(behaviour)::{
            value: behaviour.execute(this());
            future#resolve(value);
            value
          }
      });
      msg.setFuture(actor(future.new()));
      msg
    };

    send(msg)::{
      .send(msg);
      ‘ return placeholder actor instead of void ‘
      msg.getFuture()
    }
})
```

Future listener behaviour The `futureListener` object describes the behaviour of a listener actor, whose single purpose is to send an `invokeWhen` message with the resolved value as soon as it is computed (i.e. as soon as the future is resolved).

```
futureListener: root.object({
    id: void;
    reference: void;
    cloning.new(anId, aReference)::{ id:=anId; reference:=aReference };
    notify(content)::reference#invokeWhen(id, [content])
})
```

Future behaviour Finally, the `future` object defines the behaviour of a non-blocking future actor (i.e. a placeholder which gets resolved eventually). The `resolved` slot is used to hold the computed value, with a `void` value indicating that the future has not yet been resolved. The key method is `resolve(content)`, which is called right after the value of the computation is known. This method notifies all listener actors (which have previously subscribed to the future actor) of the availability of the result.

Furthermore, the placeholder actor stores every asynchronous message it receives until the future is resolved. If the resolution is also an actor, the placeholder forwards all stored messages to the resolution, and acts as a transparent proxy for all messages coming in afterwards (This is similar to the routing actor described in section 3.4.2.4).

```
future: root.object({
    resolved: void;
    subscribers: vector.new();

    cloning.new()::{subscribers:=vector.new(); resolved:=void};

    subscribe(anActor)::{
      subscribers.add(anActor);
      if(not(is_void(resolved)), {
         anActor#notify(resolved)
      })
    };

    resolve(content)::{
      ‘ notify all subscribers of the value of the resolution ‘
      subscribers.iterate(el#notify(content));
      resolved:=content;
      ‘ forward all stored messages to the resolution ‘
      inbox.asVector().iterate({
        msg: el;
        forward(msg)
      })
    };

    forward(msg)::{
      if(and(is_actor(resolved),
             ‘ messages understood by this actor must not be forwarded ‘
             not(this().containsBehaviour(msg.getName()))), {
        inbox.delete(msg);
        msg.setTarget(resolved);
        outbox.add(msg)
      })
    };

    in(msg)::{
      if(not(is_void(resolved)), {
        forward(msg)
      })
    }
})
```

3.5.2 Due-blocks

AmbientTalk's default message sending policy guarantees eventual delivery of every outgoing message. Undeliverable messages are kept in the outbox of the sender until successful delivery, possibly for an unlimited amount of time. Due-blocks are an AmbientTalk language extension that allows the programmer to alter this policy by tagging outgoing messages with a deadline. Messages that are not sent before their deadline are removed from the sender's outbox and a handler message is sent instead, with the timed out message as an argument, in order to deal with the timeout. The use of due-blocks in AmbientTalk is illustrated by the following example:

3.5.2.1 Example

In the following example, an actor's behaviour is extended with the due-blocks behaviour. The programmer uses the `due` construct to express that the newly created actor is given 10 seconds to authentice to a known authentication actor. If the `auth` message cannot be sent before this deadline, the actor is notified of the failure, by sending it the `timeout` message with the `auth` message as an argument.

```
{
  actorBhv:: object({
     authenticator: ... ‘ reference to authentication actor ‘
     loginTimeout: 10000;
     user: "john";
     pass: "doe";

     login():: { authenticator#auth(user, pass) };

     timeout(msg):: { display("Sending message '", msg.getName(),
                              "' timed out", eoln) };

     ‘ the init() message is sent automatically to ‘
     ‘ every actor at initialization ‘
     init():: {
        ‘ the third argument creates a first−class message without ‘
        ‘ sending it ‘
        due(loginTimeout, { login() }, thisActor()#timeout)
      }
    });

    extBhv: extendWithExpiryCheckingBehaviour(
                  extendWithDueBehaviour(actorBhv));
    act: actor(extBhv)
}
```

3.5.2.2 Reflective implementation

The due language construct comprises two methods:

- `extendWithDueBehaviour` extends a given behaviour with the `due` method.
- `extendWithExpiryCheckingBehaviour` extends the behaviour of an actor to continuously check its inbox and outbox for expired messages. If such messages are found, the corresponding handler message is sent.

Due-block extending behaviour `extendWithDueBehaviour` extends a given actor behaviour with the `due` method, which takes three arguments: a deadline relative to the current time, the block of code to execute and the handler message to send for every message whose deadline is not met. Whenever the `due` method is called, the timeout value and the handler message are stored in two data slots. The value of these data slots is used by `createMessage` to tag every message created inside the block of code of the due-block with an extra deadline and a “complaint address”. When `dueTimeout` is `void`, `createMessage` knows that the message was created outside of a due-block. Additionaly, saving and restoring the `dueTimeout` and `dueHandlerMsg` slots inside the `due` method allows for nested due-blocks.

```
extendWithDueBehaviour(bhv) :: bhv.extend({
    dueTimeout: void;
    dueHandlerMsg: void;

    due(deadline, body(), handlerMsg) :: {
      tmpTimeout: dueTimeout;
      tmpHandler: dueHandlerMsg;
      dueTimeout := deadline;
      dueHandlerMsg := handlerMsg;
      value: body();
      dueTimeout := tmpTimeout;
      dueHandlerMsg := tmpHandler;
      value
    };

    createMessage(aSource, aTarget, aName, anArglist):: {
        ‘ create regular message ‘
        msg: .createMessage(aSource, aTarget, aName, anArglist);
        ‘ check if message is sent from inside a due−block ‘
        if(!is_void(dueTimeout),
           { msg.extend({
                  deadline :: time() + dueTimeout;
                  handlerActor :: dueHandlerMsg.getSource();
                  handlerName :: dueHandlerMsg.getName()
              }) },
           msg)
    }
})
```

Expiry checking behaviour The expiry checking behaviour registers the actor with a "ticker" actor that sends `notify` messages at regular time intervals. At every notification, the actor iterates over its inbox and outbox to remove every message that has not been sent before its deadline and to send a handler message instead. Checking the inbox is required in case the expiry checking behaviour is added to e.g. a future, because a future typically stores incoming messages in its inbox until it is resolved (see section 3.5.1).

```
extendWithExpiryCheckingBehaviour(bhv) :: bhv.extend({
    pollInterval: 1000; ‘ in milliseconds ‘

    init():: {
      .init();
      root.createTickerActor(pollInterval)#register(thisActor()#notify)
    };

    notify():: {
      timedOut(msg, mbox):: {
        if(has_slot(msg, "deadline"), ‘ only check tagged messages ‘
           if(time() > msg.deadline,
             { ‘ send handler message to the complaint address ‘
               ‘ with the timed out message as unique argument ‘
               send(createMessage(thisActor(), msg.handlerActor,
                                  msg.handlerName, [ msg ]));
               true },
             false),
           false)
      };

      outbox.removeIf(timedOut(msg, outbox));
      inbox.removeIf(timedOut(msg, inbox))
    }
})
```

3.5.3 Discussion

We have seen throughout this section that AmbientTalk language extensions are typically implemented following a set of “patterns”. First, the `send` method is often overridden in combination with the `createMessage` method; this allows the programmer to extend “regular” messages with additional behaviour that is relevant for the language extension. Secondly, language extensions often employ mailbox observers to execute custom code whenever a message is added to a mailbox. Finally, most language extensions contain one or more methods that form the “public interface” to the application programmer, such as the `when` construct for futures and the `due` construct for due-blocks.

3.6 Conclusion

This chapter introduced a relatively new programming paradigm, called the *ambient-oriented programming paradigm*. This paradigm is especially geared towards devices communicating in open mobile networks. First, the characteristics of mobile networks were analyzed, in comparison to stationary networks. Next, these characteristics were used to define a set of criteria for ambient-oriented programming languages. Subsequently, an ambient-oriented programming language called *AmbientTalk* was introduced; special attention was paid to its meta-object protocol, which allows to implement new language constructs reflectively. Finally, two such constructs (non-blocking futures and due-blocks) were analyzed in detail.

The next chapter returns to the AmbientTalk MOP by evaluating its weaknesses and the problems assiociated with them. It also assesses the applicability of the design principles presented in chapter 2 in the context of the ambient-oriented programming paradigm.

4 Evaluating AmbientTalk's Meta-Object Protocol

This chapter evaluates AmbientTalk's meta-object protocol, as described in section 3.4.2.5. It focuses on the shortcomings of the MOP, such as a lack of separation between the language's base- and meta-level and the absence of a high-level mechanism to compose language mixins. Next, the meta-level architectures and design principles described in chapter 2 are evaluated in the context of ambient-oriented programming; their applicability and shortcomings are assessed in the light of implementing a reflective architecture for an ambient-oriented programming language.

4.1 Evaluation of AmbientTalk's meta-object protocol

While AmbientTalk provides support for reflective operations, its current meta-object protocol (see section 3.4.2.5) exhibits a series of shortcomings, as detailed below.

4.1.1 Lack of stratification and encapsulation

Recall from section 3.4.2.5 that the state and behaviour of AmbientTalk actors is represented by an (internal) passive object *p*. The meta-object protocol consists of a set of methods and fields defined directly in *p*, or "inherited" indirectly from the root object, which contains the default implementations of every reflective method. Hence, there is no separation between the base- and the meta-level—both exists side by side in the same scope. This violates the principle of stratification of a mirror-based design, which dictates that there must be a clear separation between the base-level and the meta-level of a reflective language (see section 2.3).

We illustrate the lack of stratification in figure 4.1, which depicts the example described in section 3.5.1.1. The root object contains base-level methods (`sin(x)`, `display@args`) and fields (`facActor`), as well as meta-level methods (the default implementations of `process(msg)`, `send(msg)`, ...) and fields (the reified mailboxes: `inbox`, ...). The root object is extended with the desired base-level behaviour (`printFac(i)`). Afterwards, the object that describes the actor's behaviour is itself extended with the future-enabling (meta-level) state (`whenBlocks`, `newId`) and behaviour (`invokeWhen(andId, content)`, `createMessage@args`, `send(msg)`, `when(aFuture, code(content))`—the `when` method is special because it is a meta-level method that must be "exported" to the base-level so the programmer can use it in base-level programs). Such methods are AmbientTalk's *language constructs* (see section 3.5). Finally, the resulting passive object is "wrapped" in an actor.

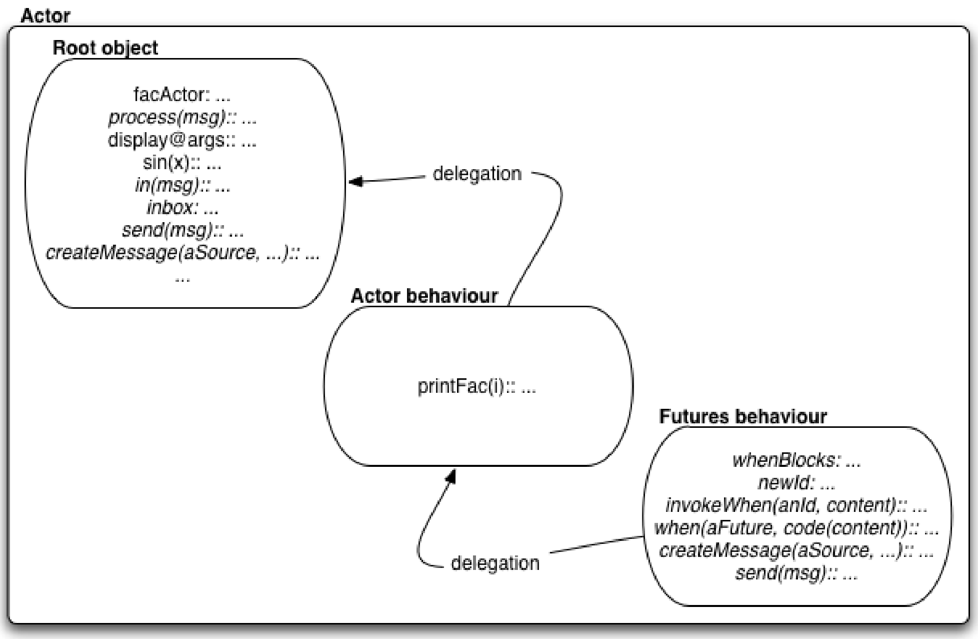


Figure 4.1: Violation of the stratification principle in AmbientTalk's MOP

Beyond the fact that it does not respect the stratification principle, this approach has multiple drawbacks:

1. It exposes the base-level programmer to unnecessary meta-level details.
2. The unwary programmer risks accidentally overriding/shadowing meta-level

methods/fields by defining a base-level method/field with an identical name as an existing meta-level method/field. For example, the programmer might define a `process(request)` method to process, say, incoming HTTP requests. This method would then be used by the interpreter to process every incoming message, instead of the `process(msg)` method defined in the root object. This could lead to errors that are undetectable to the programmer who is unaware of the meta-level architecture of AmbientTalk.

3. Similarly, meta-level programmers risk accidentally overriding/shadowing base-level methods/fields by defining a meta-level method/field with an identical name as an existing base-level method/field. For example, when we consider the expiry checking meta-behaviour defined in section 3.5.2, the `notify()` meta-method might override an actor's base-level `notify()` method which, e.g. notifies all surrounding "chat" actors of the actor's status message.

For the reasons stated above, we argue that the meta-level functionality should be clearly separated from the base-level functionality, because it is impossible to require that the base-level programmer knows the interfaces of all possible meta-level programs, and vice versa.

Furthermore, base- and meta-level implementation details, such as private fields, should be encapsulated in separate entities, such that both are isolated from one another. This corresponds to the principle of encapsulation of a mirror-based design, as defined in 2.3. Neither the stratification principle nor the encapsulation principle are supported by AmbientTalk's MOP.

4.1.2 Ad hoc composition

In AmbientTalk, language extensions such as non-blocking futures and due-blocks (see sections 3.5.1 and 3.5.2) are typically implemented by overriding a set of meta-methods (e.g. `send`, `createMessage`, `process`, ...). These meta-methods—and possibly additional state—are then mixed in the passive object that represents the actor's state and behaviour. Technically, mixing in state and behaviour is achieved by extending the current object with the code that defines the language extension (this is done by the `extendWithFuturesBehaviour`, `extendWithExpiryCheckingBehaviour` and `extendWithDueBehaviour` methods in the case of futures and due-blocks). Extending an actor with multiple language mixins is achieved by mixing in the code of all extensions sequentially. For example, suppose the programmer wants to create an actor that supports non-blocking futures (and the `when` construct) and failure handling using the `due` construct. This is achieved by the following code, where `bhv` is the actor's base-level behaviour.

```
bhv: object(...);
futBhv: extendWithFuturesBehaviour(bhv);
dueFutBhv: extendWithExpiryCheckingBehaviour(
                        extendWithDueBehaviour(futBhv));
a: actor(dueFutBhv)
```

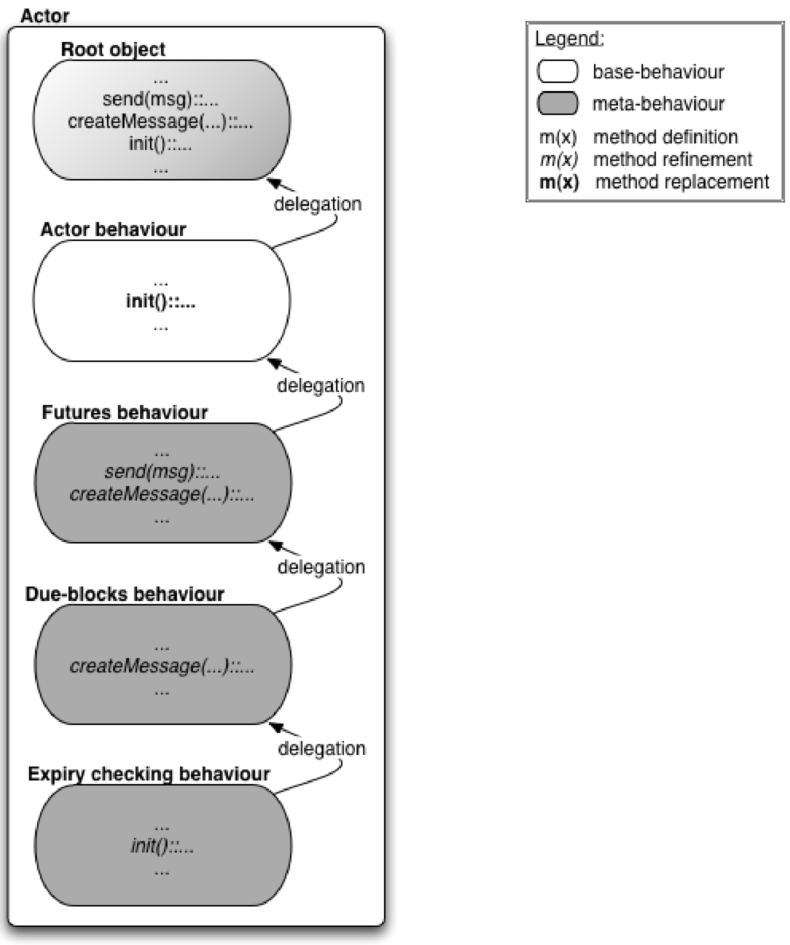


Figure 4.2: Applying futures and failure handling mixins to an actor's behaviour

The resulting actor is depicted in figure 4.2. This example illustrates that different language mixins are combined using *ad hoc composition*—i.e. without a structured high-level composition mechanism; the programmer might as well have reversed the order in which the different language extensions are mixed in. However, we claim that ad hoc composition is not sufficient and can lead to unexpected results.

To support our claim, we must first briefly recapitulate the notion of overriding in object-oriented languages. In Pic%, a method in a child object is said to *override* a method in its parent object if the two methods have the same name[1]. More precisely, there are two different ways to interpret the process of overriding [Bud01]:

- A method *replacement* totally overwrites the method in the parent object during execution; the code of the parent object is never executed when its child object is manipulated.
- A method *refinement* includes, as part of its body, the execution of the method "inherited" from the parent (through delegation). Hence, the behaviour of the parent object is preserved and augmented.

Back to the case of ad hoc composition, suppose that two language extensions m_1 and m_2 are to be mixed in an actor's base-level object. The implementation of m_1 replaces the `send` and `process` methods, while the implementation of m_2 refines the `send` and `createMessage` methods. Using ad hoc composition, the programmer mixes in both m_1's and m_2's code. However, the order in which m_1 and m_2 are mixed in influences the correctness of the result:

1. If the programmer mixes in m_1 before m_2, the result is correct, because the `send` method of both m_1 and m_2 is executed.
2. If the programmer mixes in m_2 before m_1, the result is incorrect, because the `send` method refined in m_2 is completely overwritten in m_1.

Similarly, combining two language mixins which both replace the same meta-level method (e.g. `process`) always leads to an incorrect result, because the behaviour of one of both methods is completely overwritten[2].

Additionally, suppose a language extension m depends on another language extension m' for proper operation. With ad hoc composition, the responsibility of mixing in m' before m is left to the programmer. Therefore, we advocate the need for a high-level mechanism for combining language extensions that performs the necessary checks and prevents the programmer from combining language extensions erroneously.

4.1.3 Partly closed service discovery protocol

In AmbientTalk, the service discovery protocol is only made partly available to the programmer through four native mailboxes (`provided`, `required`, `joinBox` and

[1]Being a dynamically typed language, Pic% ignores the concept of type signatures, and hence only takes the name of the method into account.

[2]The result can be correct in some rare cases, when the programmer intentionally replaces a method to disable specific behaviour from the parent. However, the programmer should be fully aware of the possible incompatibility.

`disjoinBox`). It works by matching every actor *a* to other actors by testing the interface required by *a* against the interfaces provided by the other actors. Interfaces are defined as a set of strings and testing interfaces against each other is done using string comparisons (see section 3.4.2.3).

Fully opening the implementation of the service discovery protocol would allow the programmer to define his own comparator methods, thereby allowing fine-grained control over the matching process [CDMM05]. For example, the programmer can use custom comparator methods to solve version conflicts. Suppose an actor *a* requires `FooService version 1` and an actor *b* provides `FooService version 2`, then a custom comparator could still match *a* to *b* if it is known that version 2 of `FooService` is compatible with version 1; if both versions are not compatible, the comparator might still match both actors, by providing *a* with an in-between proxy actor that converts messages from *a* in a format *b* understands and forwards them transparently to *b*.

4.2 Evaluation of the ABCL/R meta-level architecture

This section evaluates ABCL/R's meta-level architecture, as covered in section 2.4.2. It assesses in how far the approach taken in ABCL/R to build an open implementation is applicable to the ambient actor model of AmbientTalk (see section 3.4).

ABCL/R [WY88] adds reflective capabilities to the actor-based language ABCL/1 [YBS86]. Every meta-object $\uparrow x$ reifies structural base-level entities of its denotation (i.e. base-object) x such as state and behaviour, the message queue and the sequential evaluator used to process incoming messages (see section 2.4.2.2). The state and behaviour of a denotation x roughly corresponds to the part of an actor's passive object that describes base-level behaviour in AmbientTalk, while the message queue corresponds to an actor's inbox in AmbientTalk.

Behavioural intercession is also supported in ABCL/R: every object can specify a custom meta-object, thereby overriding default behaviour. An object x and its meta-object $\uparrow x$ support natural concurrency (see section 2.4.2.2), meaning that $\uparrow x$ can accept messages while x is performing a computation. Hence, as the example code of section 2.4.2.2 shows, an object can change x's methods through $\uparrow x$ while x is concurrently performing a computation. One drawback of this approach is that it introduces possible race conditions, for example if a method m is removed from the behaviour of x through $\uparrow x$ while x is still executing m.

By separating base- and meta-level behaviour in a *denotation* and a *meta-object*, ABCL/R supports the principles of stratification and encapsulation of a mirror-based design (see section 2.3). However, the fact that the programmer can specify at object creation which meta-object to use to reflect upon the new object partly breaks the stratification principle.

Furthermore, ABCL/R is not a distributed language: it does not support the distribution of objects among several machines. Hence—contrarily to AmbientTalk—it does not take into account issues such as network failure, object referencing, ... As a matter of fact, a distributed implementation of ABCL/1 happens to exist [BdR88], but it is designed for stationary networks and has no reflective implementation. Also, the `outbox`, `sentbox` and `rcvbox` used to implement full reification of communication traces in AmbientTalk have no counterpart in ABCL/R. Finally, ABCL/R lacks a high-level composition mechanism to combine language extensions.

4.3 Applicability of a mirror-based design

Recall from section 2.3 that mirror-based meta-level architectures must adhere to the principles of *stratification* (i.e. separating base- and meta-level operations in distinct entities to, amongst others, avoid name clashes between base- and meta-level slots) and *encapsulation* (meaning that base- and meta-level implementation details must be isolated from one another). Because of these desirable properties, we will use a mirror-based design as a starting point for designing a proper MOP for AmbientTalk (see chapter 5).

However, mirror-based designs do not address the issues related to actor-based programming languages, such as, for example, concurrency between the base-level and the meta-level of the language. Equally challenging is the fact that the question of how to support behavioural intercession in a mirror-based architecture remains unanswered [BU04]. However, since AmbientTalk heavily relies on behavioural intercession for implementing language extensions (see section 3.5), this issue will have to be addressed. Lastly, mirror-based designs offer no provisions for combining meta-level behaviour, other than ad hoc composition.

4.4 Conclusion

In this chapter we have reviewed AmbientTalk's current meta-object protocol, mainly focusing on its shortcomings, i.e. the total lack of stratification and encapsulation, the ad hoc mixin composition and the closed service discovery protocol. Subsequently, we have evaluated existing approaches to building reflective object-oriented architectures, with a view to implementing a proper meta-object protocol for AmbientTalk.

In the next chapter, we proceed with the actual implementation of a mirror-based MOP for AmbientTalk, taking into account the various observations made throughout this chapter and addressing the challenges described in section 4.3. The problem of ad hoc composition will be addressed in chapter 6.

5 A Mirror-based Meta-Object Protocol for Active Objects

This chapter addresses the problem of the lack of stratification and encapsulation in the AmbientTalk meta-object protocol. To this end, the mirror-based design principles (see section 2.3) are applied to AmbientTalk's meta-architecture, and the feasibility of introducing the concepts of distribution and behavioural intercession in a mirror-based design are assessed. Finally, the mirror-based MOP is evaluated in comparison to the previous MOP, and validated by reimplementing the language extensions presented in section 3.5, so that they comply with the new MOP.

5.1 Design of the mirror-based MOP

Recall from section 3.4.2.5 that AmbientTalk's MOP consists of a number of fields reifying base-level behaviour (e.g. mailboxes), and a number of protocols (i.e. orchestrated chains of method sends) describing the interactions between meta-level methods to achieve a certain result (e.g. creating a message, processing a message, acting upon the reception of a message).

Recall also from section 4 that there is no separation between the base-level and the meta-level of the AmbientTalk meta-object protocol (i.e. no stratification and encapsulation). This can lead to unexpected (erroneous) behaviour, such as when the programmer who is unaware of the meta-level architecture accidentally replaces meta-level behaviour.

Therefore we propose to implement the principle of stratification in the AmbientTalk MOP using a design based on mirrors (see section 2.3). The idea is to separate the base-level and the meta-level of every actor in two distinct passive objects: a base-level object describing all base-level behaviour (i.e. describing the external problem domain, e.g. an instant messaging service[1]), and a meta-level object describing all meta-level behaviour (i.e. describing the actor itself, e.g. message sending and processing). A sketch of the current design versus our proposed design is provided in figure 5.1. Throughout the remainder of this document, we will consistently refer to the former as the *flat* MOP and the latter as the *mirror-based* MOP.

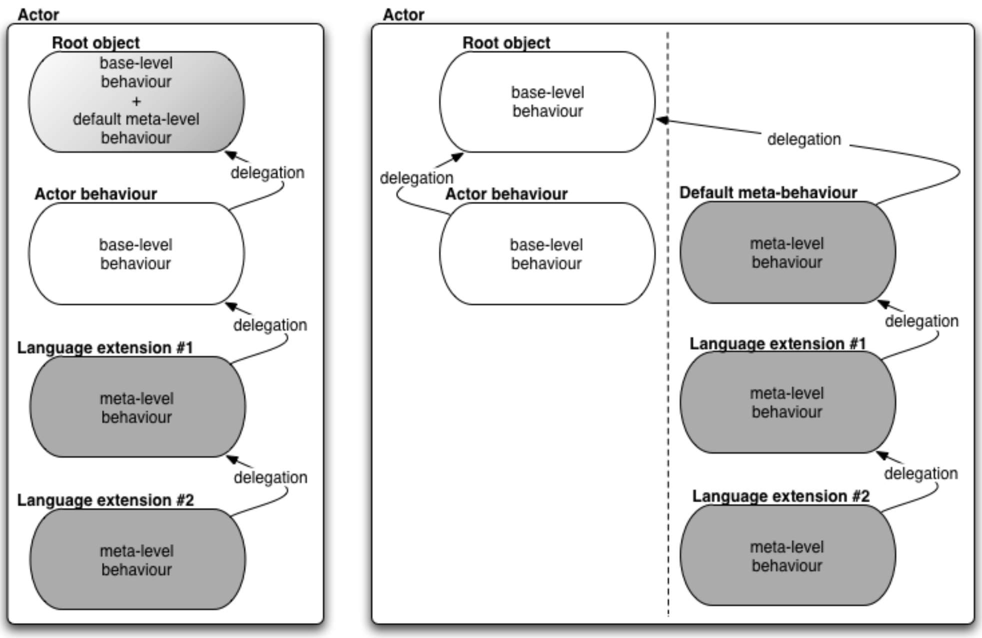


Figure 5.1: Sketch of AmbientTalk's base- and meta-level architecture: current versus proposed design

What is important to note is that base- and meta-level behaviour are stored in the same actor (instead of e.g. using a design with separate base- and meta-level actors), thereby avoiding complex synchronization schemes between the base-level and the meta-level. The possibility of race conditions, such as in ABCL/R (as explained in section 4.2) is also avoided. This design implies that meta-level messages use the same set of mailboxes as base-level messages. Hence, a mechanism is required to differentiate between both kinds of messages. This is described in more detail in section 5.1.2. The following sections zoom in on specific aspects of the design of our mirror-based MOP.

[1]An implementation of an AmbientTalk chat service is provided by Dedecker et al. [DVM$^+$06].

5.1.1 Applying the principles of stratification and encapsulation

To apply the *stratification* and *encapsulation* principles dictated by mirror-based designs, the idea is to extract all meta-level behaviour from the root object and separate it into a passive object that describes the default meta-behaviour of an actor. The programmer is then given two options to create a new actor:

1. If the programmer specifies only a base-level object at actor creation time, the object describing the default meta-level actor behaviour is used. For example, the programmer would write `actor(facBhv)` to create a "fac" actor behaving identically to the one described in section 3.5.1.

2. If the programmer specifies both a base-level and a meta-level object at creation time, the provided meta-level object is used instead of the default meta-level object. Note that any passive object implementing the required meta-level behaviour can be specified. For example, the programmer wishing to create a "fac" actor that logs all sent messages, would invoke the `actor` native with a second argument, namely a `logBhv` meta-object that describes the logging meta-behaviour. To create such a meta-object, the programmer can call the `createActorMetaBehaviour()` method, which returns a clone of the meta-object that describes the default meta-behaviour of an actor. The following code illustrates this.

```
facBhv: object(...);
logBhv: createActorMetaBehaviour().extend({
    send(msg):: { .send(msg);
                  display("Sent: ", msg.getName()) }
});
act: actor(facBhv, logBhv)
```

To prevent race conditions, a deep copy of both the base- and meta-level objects is taken in both cases (see also section 3.4). This way, no actor ever shares its base- or meta-object with another actor.

Additionally, the fact that every actor is sent the `init()` message at initialization is transposed to the mirror-based MOP by sending every newly created actor both a meta-level and base-level (in this order) `init()` message. The following section explains how one can send base- and meta-level messages to an actor.

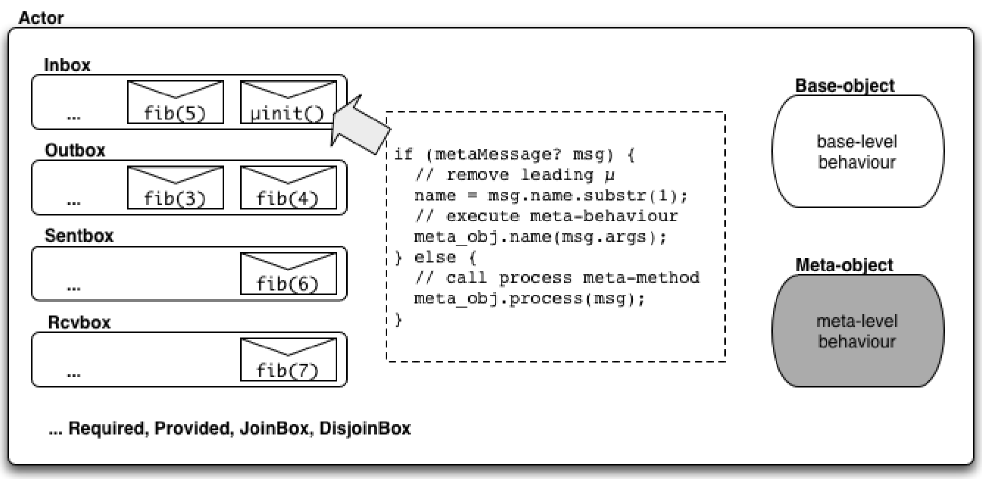
5.1.2 Sending and processing base- and meta-level messages

As a consequence of our single-actor design, base- and meta-level messages share the same set of primitive mailboxes. Therefore, a mechanism to differentiate between both kinds is required. We achieve this by preceding the name of every meta-level message with the μ character. The name of base-level messages is unchanged. For example, to send an actor `a` the `process(aMsg)` meta-message, one writes: `a#μprocess(aMsg)`.

The thread that sequentially processes messages in the inbox analyzes the name of every message and dispatches to the correct behaviour accordingly:

- For meta-level messages, the method corresponding to the message's name is looked up in the actor's meta-object and called with the message's arguments.
- For base-level messages, the `process(msg)` method of the actor's meta-object is looked up and called with the message as an argument.

Hence, meta-level messages are processed directly, instead of e.g. by a meta-meta-level `process(aMetaMessage)` method. This is a direct consequence of the absence of an *infinite reflective tower of meta-objects* in our design. Section 5.3.1 elaborates on this limitation. Meanwhile, the processing of base- and meta-level messages is depicted and explained in pseudo-code in figure 5.2 and exemplified by figure 5.3, which clearly depicts the differences between both message processing mechanisms (the meaning of the `<implicit shift>` arrows in figure 5.3 will become clear in the following section).



```
if (metaMessage? msg) {
  // remove leading μ
  name = msg.name.substr(1);
  // execute meta-behaviour
  meta_obj.name(msg.args);
} else {
  // call process meta-method
  meta_obj.process(msg);
}
```

Figure 5.2: Processing base- and meta-level messages in AmbientTalk's mirror-based MOP (1)

5.1.3 Shifting between base- and meta-level

The programmer is given two ways to shift from the base-level of an actor to its meta-level: either explicitly by requesting the corresponding object, or implicitly by exporting meta-methods to the base-level.

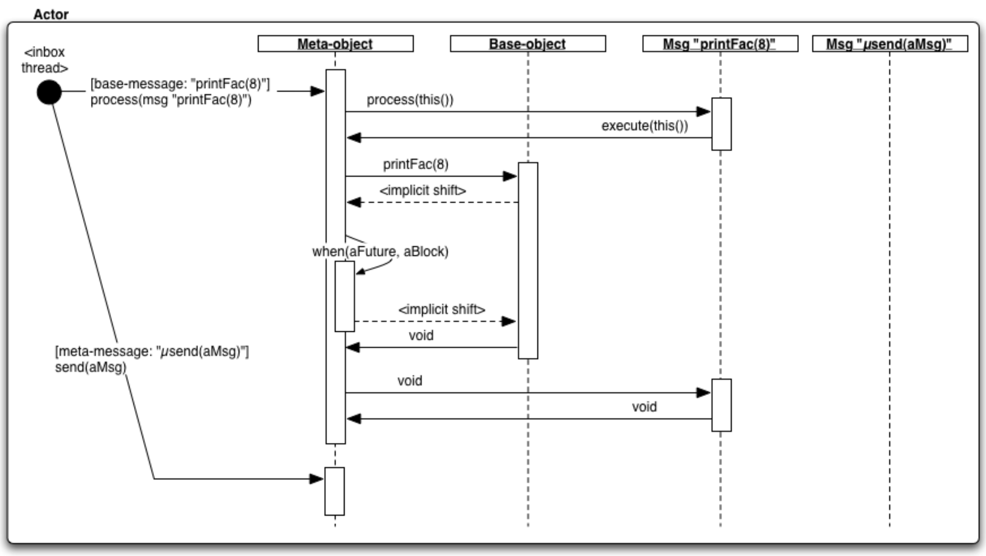


Figure 5.3: Processing base- and meta-level messages in AmbientTalk's mirror-based MOP (2)

5.1.3.1 Shifting explicitly

Shifting from within the base-object of an actor to its meta-object is supported through the `metaBehaviour()` native, which returns the actor's meta-object. Similarly, every actor's meta-object contains the `baseBehaviour()` native, which returns the actor's base-object. Note that neither of the natives needs to return a deep copy of the base-/meta-object, because the resulting objects stay confined to the actor's boundaries.

5.1.3.2 Shifting implicitly by exporting meta-behaviour

Most AmbientTalk language extensions (see section 3.5) provide a public interface which must be available to the base-level programmer. These interfaces consist of one or more meta-level methods that are explicitly used at the base-level. Examples include the `when` construct for futures and the `due` construct for due-blocks. Hence, a mechanism is needed to somehow *export* such methods from the meta-level (where they are defined) to the base-level (where they are used).

While in the flat MOP this is not an issue (since the meta-level methods of the public interface coexist side by side with the base-level methods), in the mirror-based MOP one wants to avoid having to explicitly query an actor for its meta-object every time a method of the public interface must be called. Hence, one wants to avoid the following coding pattern:

```
facBhv:: object({
    printFac(i):: {
      metaBehaviour().when(facActor#fac(i-1),
                           display("fac(", i, ") = ",
                                   content * i, eoln))
    }
});
```

Therefore, the meta-programmer must be provided with a mechanism to export a language extension's meta-level methods as a public interface to the base-level object. However, such methods must still be evaluated in the context of the meta-object. Hence, every exported method must be wrapped in a *closure* (i.e. a pair containing a method and an evaluation context for the body of the method) that contains a reference to the meta-object. This is depicted in figure 5.4. For clarity reasons, the base-level and the meta-level are each depicted as one single object instead of a chain of objects delegating to each other. With implicit shifting (using exported methods), the above code is rewritten as follows:

```
facBhv:: object({
  printFac(i):: {
    when(facActor#fac(i-1),   ` executed in the context of `
         display(...))        ` the actor's meta-object     `
  }
});
futuresBhv:: createActorMetaBehaviour().extend({
  init():: {
    .init();
    export(when)  ` export meta-method to base-object `
  };

  when(aFuture, code(content)):: { ... }
});
act: actor(facBhv, futuresBhv)
```

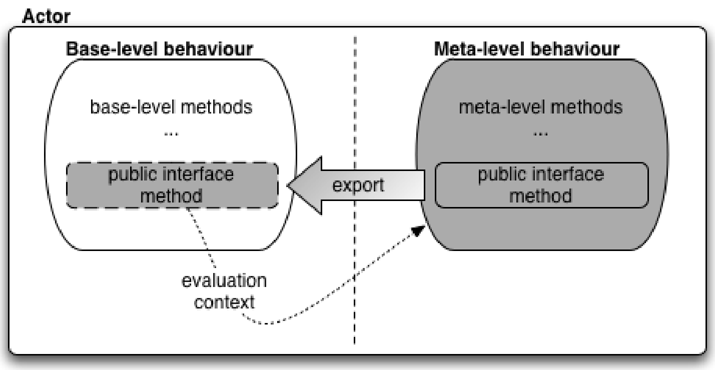


Figure 5.4: Exporting meta-level behaviour to the base-level

5.1.3.3 Natural synchronization

Due to the fact that an actor's base-object and meta-object are stored in the same actor, synchronization between the base-level and the meta-level occurs naturally. Indeed, the base-object and the meta-object of an actor send synchronous messages to one another, blocking until each method call returns. This coincides well with the semantics of shifting from the base-level to the meta-level and vice versa, which are also synchronous operations. The sequence diagram in figure 5.3 illustrates this natural synchronization.

5.1.4 Dynamically changing an actor's base- and meta-behaviour

The programmer can dynamically change an actor's base- and meta-behaviour by sending it the μ`become(aBaseObj)` and μ`metaBecome(aMetaObj)` message, respectively. All base-level (resp. meta-level) messages following a `become` (resp. `metaBecome`) are processed using the new base-object (resp. meta-object). To prevent sharing of base- or meta-objects, the arguments of `become` and `metaBecome` are deep copied. As an example, consider the following code, which dynamically adds support for logging every executed base-message:

```
a: actor(object({
    foo(i):: display("Foo: ", i, eoln)
}));
a#foo(42);
a#foo(13);
a#μmetaBecome(createActorMetaBehaviour().extend({
    process(msg)::{ display("———Start processing ", msg.getName(), eoln);
                    .process(msg);
                    display("———Processing finished") }
}));
a#foo(5)
```

Its output is given below:

```
Foo: 42
Foo: 13
---Start processing foo
Foo: 5
---Processing finished
```

Whereas the `become` native resided at the base-level in AmbientTalk's flat MOP (see section 3.4), it is implemented as a method of the actor's meta-object in the mirror-based MOP—where it conceptually belongs. The `metaBecome` native, however, conceptually belongs to an actor's meta-meta-object (i.e. a meta-object that describes the actor's meta-object). However, as already noted in section 5.1.2, one of the limitations of our mirror-based MOP is the absence of an infinite reflective tower. This forces us to assign the `metaBecome` native to the actor's meta-object.

5.2 Implementation of the mirror-based MOP

This section describes various details and considerations relative to the implementation of our mirror-based MOP for AmbientTalk.

5.2.1 Identifying meta-level behaviour

The first step at implementing a mirror-based MOP for AmbientTalk is to clearly identify the meta-level behaviour that must be separated from the root object in a distinct object. We identify four kinds of meta-level entities:

1. *Meta-level natives.* AmbientTalk extends Pic% through a set of native methods, e.g. `execute`, which executes a message in the context of a specified passive object and `become`, which dynamically changes the base-level behaviour of an actor.
2. *Meta-level methods.* A number of MOP methods such as `process` and `send` which are implemented completely in AmbientTalk code.
3. *Mailbox observers.* The methods that are called whenever a message is added to their corresponding mailbox, e.g. `in(msg)` and `sent(msg)`.
4. *Meta-level prototypes.* The prototypes for asynchronous messages and the eight native mailboxes (`inbox`, `outbox`, `sentbox`, `rcvbox`, `required`, `provided`, `joinBox` and `disjoinBox`) are implemented as AmbientTalk objects.

The meta-level methods and the mailbox observers implement behavioural reflection in AmbientTalk, while the meta-level prototypes implement structural reflection.

5.2.2 Moving meta-level elements to a separate meta-object

The second step of our implementation consists of moving all meta-level behaviour identified in the previous section to a default meta-object. This object is a regular AmbientTalk (passive) object, serving as a prototype to instantiate the meta-object of new actors. It is bound to the constant `rootActorMetaBehaviour` in the root object, and can be cloned using the `createActorMetaBehaviour()` method. The interested reader can find the full source code of the default actor meta-behaviour in appendix A.

5.2.3 Redirecting MOP message sends to the meta-object

Next, the code of the underlying evaluator must be changed to redirect MOP method sends to the actor's meta-object instead of the base-object. This comprises the `createMessage`, `process` and `send` methods, as well as the mailbox observers (`in(msg)`, `sent(msg)`, . . .). This is (partly) depicted in figure 5.3.

5.2.4 Exporting meta-behaviour

As explained in section 5.1.3.2, the mirror-based MOP introduces implicit shifts between the base-level and the meta-level of an actor, by using *exported* meta-methods. To export meta-level methods to the base-level, every meta-object is provided with an `export` meta-method. This method accepts a variable number of arguments, each being a method to export to the base-level. When exporting a meta-method, the method is looked up in the meta-object and wrapped in a closure that contains a reference to the meta-object. Next, this closure is bound in the base-object to a constant with the same name as the method. This way, the exported method is always executed in the context of the actor's meta-object. As shown in the example code of section 5.1.3.2, the `export` statement is typically placed in the body of the meta-object's `init()` method, such that every meta-method is exported exactly once[2].

5.2.5 Short-circuiting meta-behaviour

Conceptually, almost every reflective architecture consists of an infinite tower of meta-objects, each one reifying the meta-object below it. Technically, this infinity is achieved by short-circuiting meta-behaviour at a certain point, by implementing this behaviour natively in the underlying language interpreter [Mae87]. This is similar in our mirror-based MOP, except that our design does not permit an infinite reflective tower (see section 5.3.1): an actor's meta-object cannot be reflected upon by a meta-meta-object, and so on. Hence, meta-meta-behaviour must be short-circuited in AmbientTalk's language interpreter. For example, asynchronous base-level messages are created as the result of a call to the meta-object's `createMessage` method. Likewise, meta-level messages are conceptually to be created as the result of a call to the `createMessage` method of a meta-meta-object, and so forth. In the case of meta-message creation, our mirror-based MOP short-circuits this behaviour by creating meta-messages as the result of directly cloning a stored prototypical object representing a message, and instantiating it with the correct values.

As an alternative to this short-circuiting, one might consider creating meta-messages like regular base-level messages (i.e. by calling the `createMessage` method of the meta-object) and prepending their name with a μ character (see section 5.1.2). However, this will in some cases result in infinite loops. As an example of this, consider the case where an actor a refines the `createMessage` meta-method to extend every created message with a field pointing to an actor created in the body of the method (this is, for example, the case in the future-enabling meta-behaviour, as explained in section 5.4.1). Suppose also that meta-messages are created in the same manner as base-level messages, i.e. as the result of calling

[2]There are, however, no strict guarantees that the `init()` method of a meta-object will be called only once, since an actor can send a μ`init()` message to another actor anytime after initialization.

the `createMessage` method of the actor's meta-object. Whenever an asynchronous base-level message is sent from within the context of actor *a*, the `createMessage` method of *a*'s meta-object is called to create the message. This results in the creation of an extended message object in which a field is bound to a newly created actor a'. Conformably with AmbientTalk's semantics, *a* sends a' a μ`init()` and a `init()` message (see section 5.1.1). However, to create the μ`init()` message, *a* calls the `createMessage` method of its meta-object. This creates a new extended message with a field pointing to a newly created actor a'', which in turn, must be sent a μ`init()` and a `init()` message by *a*. Hence, this results in a infinite loop of message creations.

5.3 Evaluation of the mirror-based MOP

The mirror-based MOP described in this chapter was designed primarily to tackle the problems caused by the lack of stratification and encapsulation in AmbientTalk's current MOP. As described in section 4.1.1, these problems are:

1. Exposure of the base-level programmer to unnecessary meta-level details.
2. Accidental overriding/shadowing of meta-level methods/fields by the programmer who is unaware of the meta-level naming conventions.
3. Accidental overriding/shadowing of base-level methods/fields by the meta-level programmer.

The mirror-based MOP presented in this chapter addresses all of these problems by adhering to the principles of stratification and encapsulation. Since meta-level operations are isolated in a separate meta-object, the base-level programmer is not bothered with unnecessary meta-level details such as reified mailboxes and message processing. Also, neither the base-level programmer nor the meta-programmer risk accidentally overriding each other's code any more, since both the base- and the meta-level are isolated from each other in separate objects.

Furthermore, the *containment* principle of AmbientTalk, which states that no passive object is ever to be shared between two or more actors, is respected, because the actor creation native and the become/meta-become natives always take a deep copy of their arguments.

5.3.1 Limitations

As explained in sections 5.1.2 and 5.1.4, our mirror-based MOP lacks a proper tower of meta-objects, as is often found in reflective object-oriented architectures [Mae87, WY88]. Hence, an actor cannot have a meta-meta-object describing its meta-object, nor a meta-meta-meta-object describing its meta-meta-object, and so forth. This renders the implementation of e.g. the meta-become native somewhat unnatural (meta-become is conceptually a meta-meta-level operation, but in our design it has to be implemented at the meta-level). What is more, the lack of a proper reflective tower prevents language extensions from being used in the implementation of other language extensions. For example, it would be impossible to implement the due-blocks extension using, say, non-blocking futures, because all meta-meta-behaviour is short-circuited in the language interpreter (as explained in section 5.2.5). Indeed, the *due* mixin could not refine the meta-meta-level `createMessage` method to use futures, because this behaviour is short-circuited in the language interpreter and hence unavailable to the programmer.

Furthermore, our mirror-based MOP does at some points make small concessions to the principle of stratification. For example, the fact that the programmer can specify an actor's base- and meta-level behaviour simultaneously at creation time breaks the principle of stratification of mirror-based designs. It is, however, deemed necessary to support behavioural intercession in our mirror-based MOP, and stems from the fact that AmbientTalk's meta-level is extensively used to experiment with language extensions and is therefore not as easily separable from the base-level as pure mirror-based designs tend to advocate. Additionally, the implicit and explicit shifting techniques described in section 5.1.3 also break the stratification principle. However, they do so in a controllable manner, which does not break encapsulation. Implicit and explicit base-/meta-shifts are deemed necessary to support the idea of *language constructs* available to the base-level programmer. A final point where our design breaks the stratification principle is the fact that the meta-object that implements the actors' default meta-behaviour delegates to the root object, which is a base-level object (see figure 5.1). However, this is necessary because all AmbientTalk natives (`if`, `+`, ...) are located in the root object; without this link, no (useful) behaviour could be specified in an actor's meta-object.

5.4 Validation of the mirror-based MOP

To conclude this chapter, this section validates our mirror-based MOP by reimplementing the non-blocking futures (see section 3.5.1) and due-blocks (see section 3.5.2) language extensions to conform to the new MOP.

5.4.1 Non-blocking futures revisited

As explained in section 3.5.1, AmbientTalk's implementation of non-blocking futures consists of three building blocks:

1. The "future-enabling" behaviour to mix into an existing actor's behaviour, so that it supports the `when` construct and so that every asynchronous message send returns a future instead of `void`.

2. The "future listener" behaviour, used internally to signify to the calling actor that the future is resolved.

3. The “future” behaviour specifying the behaviour of the *placeholder* actor (i.e. the future itself).

In the next sections, we implement each of these building blocks to conform to the mirror-based MOP, and we compare our new implementation to the previous, “flat” implementation (see section 3.5.1).

5.4.1.1 Flat implementation

The `future` and `futureListener` behaviour define “stand-alone” actors—they do not define a block of code to be mixed in an existing behaviour—and are therefore (partly) omitted from the code of the previous implementation given below. An overview of this design is also presented in figure 5.5.

```
{
future: root.object({
    ‘ ... parts omitted for clarity reasons ... ‘

    forward(msg)::{
      if(and(is_actor(resolved),
             ‘ messages understood by this actor must not be forwarded ‘
             not(this().containsBehaviour(msg.getName()))), {
        inbox.delete(msg);
        msg.setTarget(resolved);
        outbox.add(msg)
      })
    };

});

futureListener: root.object({
    ‘ ... omitted for clarity reasons ... ‘
});

extendWithFuturesBehaviour(bhv):: bhv.extend({
    whenBlocks: vector.new();
    newId      : 1;

    invokeWhen(anId, content)::{
      whenBlocks.get(anId)(content)
    };

    when(aFuture, code(content))::{
      whenBlocks.add(code);
      aFuture#subscribe(actor(futureListener.new(newId,
                                                        thisActor())));
      newId:=newId+1;
      void
    };

    createMessage(aSource, aTarget, aName, anArglist)::{
      ‘ create a regular message ‘
```

```
   msg: .createMessage(aSource, aTarget, aName, anArglist);
   ‘ ... and extend it with future behaviour ‘
   msg.extend({
     future: void;

     getFuture()::future;
     setFuture(aFuture)::future:=aFuture;

     process(behaviour)::{
      value: behaviour.execute(this());
      future#resolve(value);
      value
     }
   });
   msg.setFuture(actor(future.new()));
   msg
  };

  send(msg)::{
   .send(msg);
   ‘ return placeholder actor instead of void ‘
   msg.getFuture()
  }
})
}
```

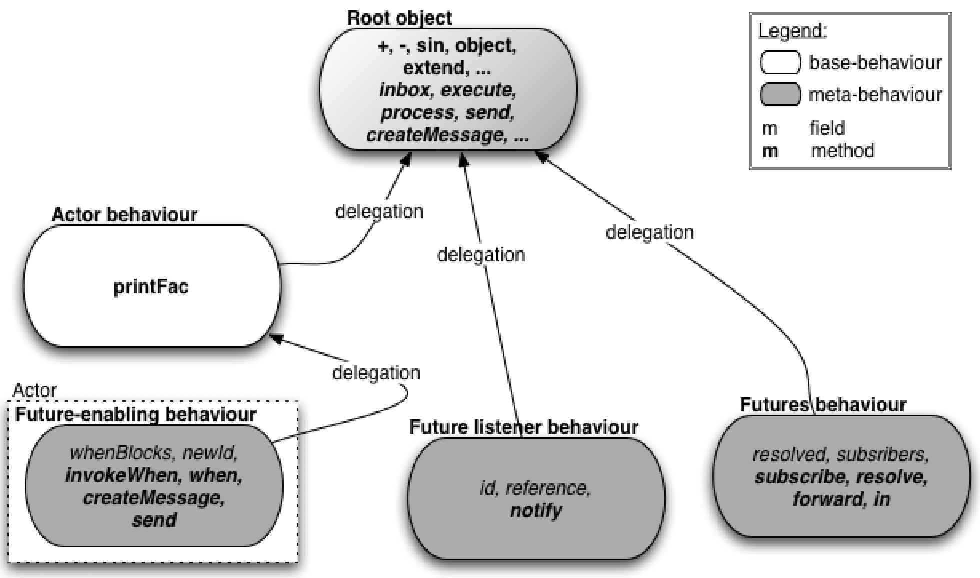


Figure 5.5: “Flat” design of non-blocking futures in AmbientTalk

Due to the lack of stratification, the above implementation presents multiple drawbacks:

1. The meta-level programmer risks accidentally overriding/shadowing base-level methods/fields. For example, if the base-level behaviour in which the future-enabling behaviour is mixed in contains an `invokeWhen(aCodeBlock(), aTimestamp)` base-level method that invokes a block of code at a specific timestamp, it will be unconditionally replaced by the `invokeWhen(anId, content)` meta-level method of the futures implementation.

2. The meta-level programmer assumes that the base-level programmer has not replaced default meta-level methods (e.g. `send`, `createMessage`, ...) with equally named base-level methods. For example, the future-enabling behaviour refines `createMessage` and therefore expects the super send (`.createMessage(...)`) to return an object representing an asynchronous message (see also section 3.4.2.5).

3. The `forward(msg)` method of the futures behaviour requires an (inelegant) ad hoc solution to check if a message is a base- or a meta-level message (and should or should not be forwarded). Even worse, suppose the base-level actor to which the future forwards messages also understands the `forward` message, then `forward` messages sent to the future will never be forwarded to that base-level actor. The following example illustrates this:

```
‘ in the context of a future—enabled actor...        ‘
email: mailApp#getMail(mailId);
‘ email is a future (i.e. a placeholder for a        ‘
‘ base—level actor representing an e—mail)           ‘
email#forward(toAddress)
‘ the base—level actor to which the future           ‘
‘ forwards messages never gets the "forward" message ‘
```

5.4.1.2 Mirror-based implementation

In this section, we reimplement AmbientTalk's non-blocking futures in order to conform to the mirror-based MOP presented in this chapter. A sketch of our mirror-based design of non-blocking futures is given in figure 5.6, while the next paragraphs detail every aspect of the mirror-based implementation.

A first observation to be made is that the code of the futures implementation belongs at the meta-level. Hence, because of the separation of base- and meta-level in the mirror-based MOP, the future-enabling code now extends a meta-object instead of a base-object. The `init()` meta-method—which, as explained in section 5.1.1, is called at actor creation time—includes the necessary `export(when)` statement to export the `when` language construct to the actor's base-level object.

Furthermore, as the behaviours of the "future listener" actor and the placeholder actor are moved to the meta-level, they now both extend the default actor meta-object (created with `createActorMetaBehaviour()`). In both cases, the

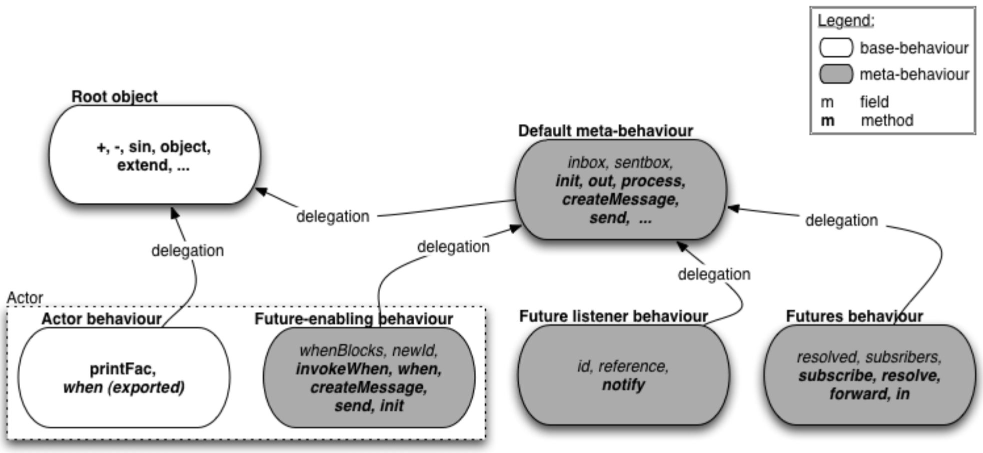

Legend:
base-behaviour
meta-behaviour
m field
m method
Root object
+, -, sin, object, extend, ...
Default meta-behaviour
inbox, sentbox, init, out, process, createMessage, send, ...
delegation
delegation
delegation
delegation
delegation
Actor
Actor behaviour
printFac, when (exported)
Future-enabling behaviour
whenBlocks, newId, invokeWhen, when, createMessage, send, init
Future listener behaviour
id, reference, notify
Futures behaviour
resolved, subsribers, subscribe, resolve, forward, in

Figure 5.6: Mirror-based design of non-blocking futures in AmbientTalk

`root` object is used to describe the base-level of the actors. Meta-level message sends between the actors are now explicitly distinguished from base-level sends and are preceded accordingly with the μ character.

```
{
  extendWithFutureMetaBehaviour(metaBhv):: metaBhv.extend({
      whenBlocks: vector.new();
      newId: 1;

      cloning.new():: {
        .new();
        whenBlocks:= vector.new()
      };

      init():: {
        .init();
        export(when)
      };

      invokeWhen(anId, content):: whenBlocks.get(anId)(content);

      when(aFuture, code(content)):: {
        whenBlocks.add(code);
        aFuture#μsubscribe(actor(root, futureListener.new(newId,
                                                    thisActor())));
        newId:= newId + 1;
        void
      };

      createMessage(aSource, aTarget, aName, anArglist):: {
        aFuture: actor(root, futuresMetaBehaviour.new());
        msg: .createMessage(aSource, aTarget, aName, anArglist);
        msg.extend({
            future: void;

            getFuture()::future;
            setFuture(aFuture)::future:=aFuture;

            process(behaviour)::{
              value: behaviour.execute(this());
              ‘ send explicit meta–message ‘
              future#μresolve(value);
              value
            }
        });
        msg.setFuture(aFuture);
        msg
      };

      send(msg):: {
        .send(msg);
        msg.getFuture()
      }

  });

  futureListener: createActorMetaBehaviour().extend({
      id: void;
      reference: void;

      cloning.new(anId, aReference):: {
        id:= anId;
        reference:= aReference
      };

      notify(content):: reference#μinvokeWhen(id, content)

  });

  futuresMetaBehaviour:: createActorMetaBehaviour().extend({
      resolved: void;
      subscribers: vector.new();

      cloning.new():: {
        .new();
        subscribers:= vector.new();
        resolved:= void
      };

      subscribe(anFutureListenerActor):: {
        subscribers.add(anFutureListenerActor);
        if(not(is_void(resolved)),
           anFutureListenerActor#μnotify(resolved)
        )
      };

      resolve(content):: {
        subscribers.iterate(el#μnotify(content));
        resolved:= content;
        inbox.asVector().iterate({
          msg: el;
          forward(msg)
        })
      };

      forward(msg):: {
        ‘ no need to test msg, it will always be ‘
        ‘ a base—level message                    ‘
        if(is_actor(resolved), {
          inbox.delete(msg);
          msg.setTarget(resolved);
          outbox.add(msg)
        })
      };

      in(msg):: if(not(is_void(resolved)), forward(msg))

  })
}
```

The example given in 3.5.1 is now rewritten as follows:

```
{
  facActor:: actor(object(
      fac(n):: if(n = 0, 1, n * fac(n—1))
  )); ‘ No custom meta—behaviour required ‘

  facBhv:: object({
      printFac(i):: {
        when(facActor#fac(i—1),
             display("fac(", i, ") = ", content * i, eoln))
      }
  });

  act: actor(facBhv,
             extendWithFutureMetaBehaviour(
                 createActorMetaBehaviour()));
  act#printFac(42)
}
```

As a result of our new design, the drawbacks of the “flat” implementation (see section 5.4.1.1) are avoided in our mirror-based MOP:

1. The meta-level programmer no longer risks accidentally overriding/shadowing base-level methods/fields, because base- and meta-level behaviour and implementation details are now separated in different objects (i.e. the stratification and encapsulation principles of mirror-based designs). For example, a base-method `send(aGreeting)` and the meta-method `send(msg)` can now exist side by side, each in their respective object.

2. Similarly, the meta-programmer can be sure that super sends in refined meta-methods always call meta-level methods.

3. Finally, the `forward(msg)` meta-method no longer requires an ad hoc solution to differentiate messages that are intended for the future from messages that are to be forwarded to the base-level actor the future resolves to. Indeed, messages addressed to the future itself are meta-messages and must therefore begin with a μ character, while messages intended for the base-actor (i.e. the resolution of the future) are base-message with no special prefix.

5.4.2 Due-blocks revisited

Recall from section 3.5.2 that AmbientTalk's due-block language extension consists of two behaviours:

1. The "due-block" behaviour extends the behaviour of an actor with the `due` language construct, allowing the programmer to put time restrictions on the validity of asynchronous messages sent in a block of code.

2. The "expiry checking" behaviour extends the behaviour of an actor so that it monitors its inbox and outbox continuously in order to remove timed out messages.

In the next sections, we implement each of these building blocks to conform to the mirror-based MOP, and we compare our new implementation to the "flat" implementation from section 3.5.2.

5.4.2.1 Flat implementation

The `extendWithDueBehaviour` and `extendWithExpiryCheckingBehaviour` define the "due-block" behaviour and the "expiry checking" behaviour, and can be mixed in an actor's behaviour jointly or separately. A recapitulation of the previous implementation is presented below.

```
{
extendWithDueBehaviour(bhv) :: bhv.extend({
    dueTimeout: void;
    dueHandlerMsg: void;

    due(deadline, body(), handlerMsg) :: {
      tmpTimeout: dueTimeout;
      tmpHandler: dueHandlerMsg;
      dueTimeout := deadline;
      dueHandlerMsg := handlerMsg;
      value: body();
      dueTimeout := tmpTimeout;
      dueHandlerMsg := tmpHandler;
      value
    };
    createMessage(aSource, aTarget, aName, anArglist):: {
      ‘ ... omitted for clarity reasons ... ‘
    }
});

extendWithExpiryCheckingBehaviour(bhv) :: bhv.extend({
    pollInterval: 1000; ‘ in milliseconds ‘

    init():: {
      .init();
      root.createTickerActor(pollInterval)#register(thisActor()#notify)
    };

    notify():: {
      ‘ ... omitted for clarity reasons ... ‘
    }
})
}
```

Similarly to the futures example, the lack of stratification can lead to accidental overriding of base-level methods. For example, the actor's behaviour might contain a `notify()` base-level method to broadcast a status message over the network. If extended with the "expiry checking" behaviour, the base-level `notify()` method would be unconditionally replaced by the meta-level method with the same name.

Additionally, due to the lack of encapsulation, the base-level programmer can freely access and/or modify meta-level fields such as `dueHandlerMsg` and `dueTimeout`.

5.4.2.2 Mirror-based implementation

Similarly to futures, the `extendWithDueBehaviour` and `extendWithExpiryCheckingBehaviour` methods specify meta-level behaviour, and should therefore extend meta-objects instead of base-objects.

Furthermore, the `init()` meta-level method of the "due-block extending" behaviour refines the `init()` method from its parent to export the `due` language construct to the base-object.

Finally, on a sidenote, the `createMessage` method is created by the higher order method `createCreateMessage(timeout, handlerMsg)`, which creates `createMessage` methods for a fixed pair of timeout/handler message values.

```
{
extendWithDueBehaviour(metaBhv):: metaBhv.extend({

    init():: {
          .init();
          export(due)
    }
    due(deadline, body(), handlerMsg):: {
          tmpCreateMessage: createMessage;
          createMessageMethod:= createCreateMessage(deadline, handlerMsg);
      value: body();
      createMessageMethod:= tmpCreateMessage;
      value
    };

    ` higher order method returning a "createMessage" method `
    createCreateMessage(timeout, handlerMsg):: {
      createMessage(aSource, aTarget, aName, anArglist):: {
               ` create regular message `
               msg: .createMessage(aSource, aTarget, aName, anArglist);
               msg.extend({
                   deadline :: time() + timeout;
                   handlerActor :: handlerMsg.getSource();
                   handlerName :: handlerMsg.getName()
               })
          }
    }

    createMessageMethod@args: .createMessage@args;
    createMessage@args:: createMessageMethod@args

});

extendWithExpiryCheckingBehaviour(metaBhv) :: metaBhv.extend({
    pollInterval: 1000; ` in milliseconds `

    init():: {
      .init();
      root.createTickerActor(pollInterval)#register(thisActor()#μnotify)
    };

    notify():: {
      timedOut(msg, mbox):: {
        if(has_slot(msg, "deadline"), ` only check tagged messages `
           if(time() > msg.deadline,
             { ` send handler message to the complaint address `
               ` with the timed out message as unique argument `
               send(createMessage(thisActor(), msg.handlerActor,
                                  msg.handlerName, [ msg ]));
               true },
             false),
           false)
      };

      outbox.removeIf(timedOut(msg, outbox));
      inbox.removeIf(timedOut(msg, inbox))
    }
})
}
```

The “log in” example given in 3.5.2 can be rewritten as follows in the mirror-based MOP:

```
{
  baseBhv:: object({
     authenticator: ... ‘ reference to authentication actor ‘
     loginTimeout: 10000;
     user: "john";
     pass: "doe";

     login():: { authenticator#auth(user, pass) };

     timeout(msg):: { display("Sending message '",
                              msg.getName(), "' timed out", eoln) };

     init():: {
       due(loginTimeout, { login() }, thisActor()#timeout)
     }
  });

  metaBhv: extendWithExpiryCheckingBehaviour(
             extendWithDueBehaviour(
               createActorMetaBehaviour()));

  act: actor(baseBhv, metaBhv)
}
```

As it was also the case with the futures implementation, the application of the stratification and encapsulation principles to our mirror-based MOP solves the issues associated with the “flat” implementation (see section 5.4.2.1). Indeed, base- and meta-level methods/fields can no longer override/shadow one another (as a result of the principle of stratification), nor can they freely access each other’s private fields (as a result of the principle of encapsulation).

5.5 Conclusion

This chapter presented the results of the design and implementation of a mirror-based MOP for AmbientTalk, a distributed actor-based programming language. This mirror-based MOP addresses some of the lacunae of the previous MOP, namely the lack of stratification and encapsulation. The lack of stratification was responsible for the possible accidental overriding of base-level behaviour by meta-level behaviour and vice versa. The lack of encapsulation, for its part, resulted in unnecessary and potentially harmful exposure of the base-level programmer to meta-level implementation details. To validate the mirror-based MOP, the implementations of non-blocking futures and due-blocks were rewritten, in compliance with the new MOP. As expected, the mirror-based MOP eliminates the issues related to the lack of stratification and encapsulation.

However, composition of meta-level behaviour is still performed in an ad hoc manner, which—as explained in section 4.1.2—can lead to erroneous behaviour when meta-behaviour is combined incorrectly. Therefore, the following chapter explores a possible solution to the *ad hoc composition problem*.

6 Language Mixin Composition

As explained in chapter 4, AmbientTalk's "flat" meta-object protocol has three major drawbacks: the lack of stratification and encapsulation, the ad hoc language mixin composition and the (partly) closed service discovery protocol. Chapter 5 addressed the first drawback by introducing a mirror-based MOP for AmbientTalk. However, as stated in section 4.1.2, mirror-based designs do not address the problem of incorrect meta-level behaviour composition. On that account, this chapter aims to implement a high-level language mixin composition mechanism to aid in creating proper meta-level mixin compositions in AmbientTalk.

6.1 Limitations of ad hoc composition

As indicated in section 4.1.2, ad hoc composition presents various limitations. This section briefly recapitulates the main disadvantages associated with ad hoc composition. Please note that throughout the remainder of this chapter, "MOP" consistently refers to the mirror-based MOP described in the previous chapter, unless stated otherwise.

6.1.1 Composition as a mere "side-effect"

In AmbientTalk's flat MOP as well as in the mirror-based MOP, language mixins are defined as regular methods that take, respectively, a base-object (see section 3.5) or a meta-object (see section 5.4) as an argument and extend it with additional behaviour—typically a combination of overridden methods, mailbox observers and private fields. Examples of such methods include the `extendWithDueBehaviour` and `extendWithFutureMetaBehaviour` methods explained in section 5.4. Hence, there is no structured way of representing language mixins in AmbientTalk and thus there is now way to discriminate between a language mixin and an ordinary method; language mixins are no more than blocks of code that extend actors' meta-objects or other language mixins.

6.1.2 Erroneous composition

As explained in section 4.1.2, ad hoc composition of language mixins can lead to incorrect results, for multiple reasons:

- A language mixin m_1 can accidentally replace one or more methods of another mixin m_2, with the effect that parts of m_2's meta-behaviour are no longer executed.
- The programmer risks inadvertently applying a mixin m more than once to the same behaviour.
- The programmer must manually examine and enforce the requirements between language mixins. If a mixin m_1 requires another mixin m_2, it is the programmer's responsibility to mix in m_2 before m_1.
- Multiple language mixins risk exporting meta-methods with identical names to an actor's base-object, such that either one of the exported methods becomes inaccessible.

6.2 Design of the mixin composition mechanism

This section describes the requirements and the design of a high-level language mixin composition mechanism for AmbientTalk.

6.2.1 Requirements

The observations made in section 6.1 result in a set of requirements for a high-level language mixin composition mechanism:

1. *A structured mixin annotation mechanism.* Language mixins should be promoted from unstructured methods to structured objects with a rich annotation mechanism to specify e.g. what other mixins a mixin requires and what meta-methods a mixin overrides.
2. *Automatic inclusion of required mixins.* The application of a language mixin m to a meta-object should trigger the automatic application of all mixins required by m, before m is applied.
3. *Automatic ordering of mixin applications.* Given a set M of mixins, the mixin composition mechanism should find an order for the mixins $m_1,...,m_n \in M$ such that:

- $\forall m_i, m_j \in M$, if m_i requires m_j, then m_j is to be applied before m_i.
- $\forall m_i, m_j \in M$, if m_i replaces a method of m_j, then m_i is to be applied before m_j.

If such an order cannot be found, the composition mechanism should fail and report which mixins prevented the mechanism from determining a correct order.

4. *Composition checks.* The composition mechanism should prevent mixins from being applied more than once to the same behaviour, and prevent two mixins m_1 and m_2 from being applied to the same passive object if both m_1 and m_2 export meta-methods with identical names.

6.2.2 Design

Based on the above requirements, we elaborate a domain-specific language for a high-level language mixin composition mechanism for AmbientTalk. The design is detailed below.

6.2.2.1 A structured mixin annotation mechanism

We solve the first requirement by introducing a proper *mixin* object to represent language mixins. Mixin objects are created using the `languageMixin` method, and contain the following information:

- A `name` field containing the name of the mixin.
- An `exported` field containing a table of names of exported methods.
- A `required` field that holds a table of other mixin objects required for proper operation.
- A `replaced` field holding a table of names of meta-methods replaced by the mixin.
- A `refined` fields that contains a table of names of meta-methods refined by the mixin.
- An `implementation` field which points to a method that will extend a given object with the behaviour of the language mixin.

For example, a simple *send tracer* mixin that refines the `send` meta-method to log all messages sent by an actor (see section 3.4.2.5 for a similar example), and requires another mixin object `fooMixin`, may be created as follows:

```
languageMixin("send tracer",
              exports(),
              requires(fooMixin),
              replaces(),
              refines(send),
              implements(bhv.extend({
                send(msg):: { display("Sending: ", msg.getName(), eoln);
                             .send(msg) }
})))
```

The result of this method call is a proper, distinguishable language mixin object.

6.2.2.2 Composing meta-behaviour

To combine different language mixin objects together, we introduce a ++ composition operator. It takes a default meta-behaviour or a composer object (see next section) as a first argument and a language mixin object as a second argument. The idea is that the programmer declares a set of mixins to combine and calls a `compose` method at the end. This is illustrated in the following example:

```
baseBehaviour: object(...);
dueMixin: languageMixin(...);
futuresMixin: languageMixin(...);
actor(baseBehaviour, compose(createActorMetaBehaviour() ++
                             dueMixin ++
                             futuresMixin))
```

6.2.2.3 Meta-behaviour composers

To support the above idea of composition, we introduce *meta-behaviour composer* objects in which we wrap a meta-object, together with a list of mixin objects to apply to it and already applied to it, and a list of methods already exported by the meta-object. Specific implementation details are described in section 6.3.

6.2.2.4 Ordering language mixins

Conceptually, mixins requiring other mixins form a direct acyclic graph in which the nodes represent the mixins and the (directed) edges represent the requirements; an edge $A \rightarrow B$ expresses the fact that mixin A requires mixin B. As an example, consider the mixin dependency graph depicted in figure 6.1.

We can apply the same principle to method replacement. Recall from section 4.1.2 that if a mixin m_1 replaces a meta-method x, and if another mixin m_2 refines the same method x, m_2 should be mixed in before m_1. Hence, we can represent method replacement as a directed graph in which the nodes correspond to the mixins and the (directed) edges represent method replacement; an edge $A \rightarrow B$

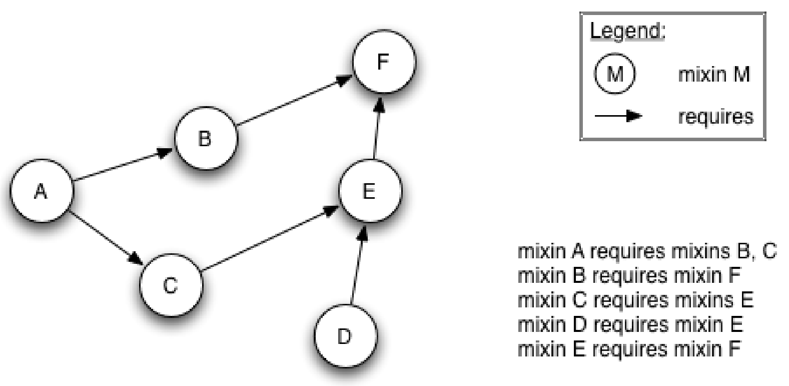


Figure 6.1: Mixin requirements as a directed acyclic graph

indicates that at least one method of mixin *A* is replaced by a method of mixin *B*. Note that such a graph can be cyclic if both *A* and *B* replace the same method. Figure 6.2 extends figure 6.1 with such new edges.

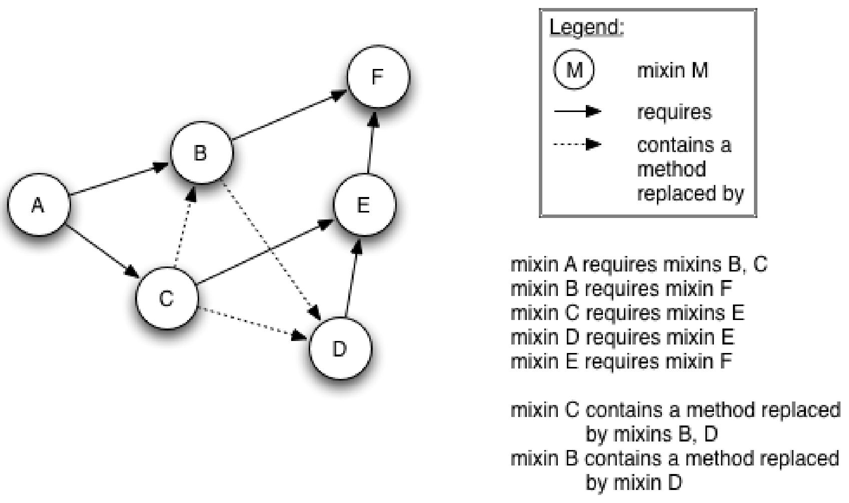

Legend:
M mixin M
requires
contains a method replaced by
A
B
C
D
E
F
mixin A requires mixins B, C
mixin B requires mixin F
mixin C requires mixins E
mixin D requires mixin E
mixin E requires mixin F
mixin C contains a method replaced by mixins B, D
mixin B contains a method replaced by mixin D

Figure 6.2: Mixin requirements and method replacement as a directed (possibly cyclic) graph

As explained in section 6.2.2.2, the programmer only needs to declare *what* mixins he wants to combine; the composition mechanism takes care of finding *how* they must be combined. Indeed, by creating a directed graph to which we add the mixins as nodes as well as the automatically inferred “requirement” and “replacements” edges between the nodes, we obtain a graph such as depicted in figure 6.2.

By topologically sorting this graph[1] (treating the "requirement" and "replacement" edges equally), we obtain a valid order in which to apply the mixins. It then suffices to apply the mixins in the reverse order of the result. For example, a possible result of topologically sorting the graph depicted in figure 6.2 is the list A,C,B,D,E,F. Hence, applying the mixins in the reverse order (F,E,D,B,C,A) yields a correct result. If the graph contains cycles, the topological sort—and hence the composition mechanism—will fail due to the mixins which form the cycles. This information can be used to report the list of conflicting mixins to the programmer. Cycles can occur e.g. if a mixin m_1 requires a mixin m_2 and at the same time m_1 replaces a method of m_2, or if both m_1 and m_2 replace the same method.

6.3 Implementation of the mixin composition mechanism

This section describes some important implementation details of our language mixin composition mechanism. The interested reader can find the full implementation in appendix B.

6.3.1 Meta-behaviour composers

Meta-behaviour composer objects are created internally using the `makeMixinComposer` method. They contain the necessary information to keep track of mixin compositions in progress, as well as the key operations of the composition mechanism.

- The `applyMixin(aMixin)` private method takes a mixin object as an argument and applies it to the wrapped meta-object after verification that the mixin has not yet been applied to the meta-object, and that the mixin does not export meta-methods with names identical to meta-methods exported by previously applied mixins. This corresponds to the fourth requirement of section 6.2.1.
- The `addMixin(aMixin)` public method "enqueues" a mixin for composition. The actual composition of all queued mixins is triggered by the `compose()` public method.
- The `compose()` public method triggers the actual composition mechanism. It computes the composition graph (see section 6.2.2.4) and topologically sorts it to find an order in which to apply the mixins. If an order is found, the mixins are applied in that order. If no order is found, no mixin is applied and an explanatory error message is produced. This corresponds to the second and third requirement.

[1] i.e. finding an order such that each node comes before all nodes to which it has edges.

6.4 Evaluation of the mixin composition mechanism

The high-level mixin composition mechanism presented in this chapter fulfills all the requirements presented in section 6.2.1.

1. It offers *a structured mixin annotation mechanism* to annotate the methods exported and overridden by a mixin as well as the mixins required by a mixin for proper operation.
2. *Automatic inclusion of required mixins.* The mixin composition mechanism automatically applies the mixins $m_1, ..., m_n$ required by a mixin m before applying m.
3. The composition mechanism *automatically orders mixin applications* such that mixins replacing methods of other mixins are applied beforehand. If such an order cannot be found, the composition mechanism fails and reports which mixins prevented the mechanism from determining a correct order.
4. The composition mechanism performs *composition checks* to prevent mixins from being applied more than once to the same behaviour, and to prevent two mixins m_1 and m_2 from being applied to the same passive object if both m_1 and m_2 export meta-methods with identical names.

6.4.1 Limitations

A first limitation of our composition mechanism is that it relies exclusively on the mixins' annotations made by the programmer. Hence, methods that are not declared in a mixin's annotation can still be replaced without a warning. However, due to the highly dynamic nature of AmbientTalk, it is doubtful if the mechanism can be extend to e.g. automatically extract this information from the mixins' implementation code.

Another limitation of our composition mechanism is its linearity. If the topological sort cannot find a linear order that satisfies every constraint, the composition mechanism will simply fail. For example, no suitable composition order can be found for a mixin *A* that refines `send` and replaces `createMessage`, and a mixin *B* that replaces `send` and refines `createMessage`. Hence, `compose(A ++ B)` will fail. To support non-linear composition, our mechanism could be extended with the possibility to explicitly resolve such conflicts. For example, writing something like

```
compose(A ++ B) with { send(msg):: { A.send(msg); B.send(msg) };
                       createMessage@args:: B.createMessage@args }
```

would permit a valid composition of *A* and *B* by explicitly resolving the conflicts created by the `send` and `createMessage` methods. This is the approach taken by *traits* [SDNB02].

6.5 Validation of the mixin composition mechanism

We now focus on the validation of our composition mechanism by reimplementing the futures and due language extensions to take advantage of the new provisions. The following code illustrates this. Note that the behaviour that maps unique IDs to closures, as required by the futures mixin, is moved to a separate mixin, such that it can also be used by other mixins that require the encoding of closures as unique integer IDs. The reason for transmitting IDs of closures instead of the closures themselves is to be found in the fact that the *containment principle* of AmbientTalk dictates that all arguments of asynchronous messages are to be deep copied (see section 3.4.2.2). Because of this, the lexical scope of every closure would also be deep copied, and side-effects such as assignments would go unnoticed in the original actor.

```
idMapperMixin:: languageMixin(
    "Closure to ID mapper",
    exports(),
    requires(),
    replaces(),
    refines(),
    implements(bhv.extend({

      id: 0;
      closures: smallmap.new();

      registerClosure(clo):: {
        id := id + 1;
        closures.put(id, clo);
        id
      };

      invokeClosure(id, args):: {
        clo: closures.get(id);
        if(not(is_void(clo)),
           clo@args)
      };

      invokeClosureOnce(id, args):: {
        invokeClosure(id, args);
        closures.delete(id)
      }
    }))
);

futuresMixin:: languageMixin(
    "Futures",
    exports(when),
    requires(idMapperMixin),
    replaces(),
    refines(init, createMessage, send),
    implements(bhv.extend({
        futureListener: createActorMetaBehaviour().extend({
            id: void;
            reference: void;

            cloning.new(anId, aReference):: ` ... `

            notify(content):: reference#μinvokeClosure(id, [content])
        });

        futuresMetaBehaviour:: createActorMetaBehaviour().extend({
            ` ... `
        });

        cloning.new():: ` ... `

        init():: ` ... `

        when(aFuture, code(content)):: {
          cloId: registerClosure(code);
          aFuture#μsubscribe(actor(root, futureListener.new(cloId,
                                                            thisActor())));
          void
        };

        createMessage(aSource, aTarget, aName, anArglist):: ` ... `

        send(msg):: ` ... `
    }))
);

dueMixin:: languageMixin(
    "Due",
    exports(due),
    requires(),
    replaces(),
    refines(init, createMessage),
    implements(bhv.extend({

        init():: ` ... `
        due(deadline, body(), handlerMsg):: ` ... `
        createCreateMessage(timeout, handlerMsg):: ` ... `

        createMessageMethod@args: .createMessage@args;
        createMessage@args:: createMessageMethod@args
    }))
);

expiryMixin:: languageMixin(
    "Expiry Checking",
    exports(),
    requires(),
    replaces(),
    refines(),
    implements(bhv.extend({
        pollInterval: 1000; ‘ in milliseconds ‘

        init():: ‘ ... ‘
        notify():: ‘ ... ‘
    }))
);

act: actor(root, compose(createActorMetaBehaviour() ++
                         futuresMixin ++
                         dueMixin ++
                         expiryMixin))
```

The output of the program is:

```
Applying mixins in order: Closure to ID mapper, Expiry Checking, Due, Futures,
```

This is, as expected, a correct result. Note that, in the above example, the `futuresMetaBehaviour` and `futureListener` objects must be encapsulated inside the `extend` block of the futures mixin, such that they are available in the scope of `when` and `createMessage`. Also note that the programmer can still create actors using ad hoc composition, e.g. when only one mixin needs to be applied.

6.6 Conclusion

This chapter tackles the second problem of AmbientTalk's "flat" MOP, namely the ad hoc composition problem. It does so by introducing a high-level language mixin composition mechanism in the form of a small domain-specific language, that is able to prevent the problems associated with ad hoc composition. The proposed solution aims to create a *declarative* layer in which the programmer only declares which language mixins to apply to a meta-object. Subsequently, the composition mechanism automatically performs the necessary checks and calculates a correct order in which to apply the mixins.

The next chapter ends this dissertation with a set of concluding remarks and pointers for future research.

7 Conclusions

As explained in chapter 1, the context of this dissertation is situated in the area of distributed actor-based languages for mobile networks. Such networks are characterized mainly by the high volatility of their participants' connections (abrupt disconnections being the rule rather than the exception in mobile networks). The observation that mobile networks are significantly different from stationary networks has led researchers to design a new paradigm for programming applications running in mobile networks. This *ambient-oriented programming paradigm* incorporates the most important characteristics of mobile networks at the very heart of its computational model [DVM$^+$06].

Being a relatively new programming paradigm, there is also a need to experiment with language constructs in ambient-oriented settings. Examples of such constructs include non-blocking futures and due-blocks. Both were presented in section 3.5. The technique chosen to implement such language constructs in AmbientTalk (our experimental language platform) is *computational reflection*. Computational reflection, and more particularly behavioural intercession, allows the programmer to tailor a particular reflective language to his specific needs, as we have extensively discussed in chapter 2.

This brings us to the goal of our research, namely the study of reflective architectures and design principles for ambient-oriented languages, and the incorporation of these reflective architectures and design principles in AmbientTalk's current meta-object protocol with the purpose of making AmbientTalk more suitable for experimenting with ambient-oriented language constructs.

7.1 Reflections on our methodology

This section reflects on the methodology adopted in our dissertation. It explains, amongst others, why we have turned ourselves to AmbientTalk to conduct our experiments, and how our contributions could be appropriate for other distributed actor-based languages.

7.1.1 Why AmbientTalk?

We have turned ourselves to AmbientTalk as the language platform to conduct our experiments because of the adequacy of the ambient-oriented paradigm to address the characteristics of mobile networks. AmbientTalk is an experimental language that is built around the ambient-oriented programming paradigm.

Furthermore, AmbientTalk is already conceived as a reflective kernel. This allows the programmer to easily adapt the language in order to experiment with new language constructs (non-blocking futures and due-blocks being two of them used extensively throughout this dissertation). Hence, because AmbientTalk is an ambient-oriented language designed as a reflective kernel, it proved to be an ideal starting point for our research.

7.1.2 Contributions

When studying AmbientTalk, we observed that its meta-level architecture lacks a number of important software engineering properties. Its most important drawbacks include the lack of stratification and encapsulation of the base-level and the meta-level of the language, and the absence of a high-level language mixin composition mechanism to combine experimental language constructs. The first limitation raises the question of how to protect the base-level and the meta-level of a language from accidentally overriding each other's behaviour, while the second limitation requires the meta-level programmer to manually ensure that multiple language extensions do not accidentally override each other's behaviour when combined together.

This dissertation endeavours to address these limitations. Therefore, we started with an analysis of current state of the art object-oriented reflective techniques (e.g. meta-object protocols) and design principles (e.g. mirrors) and assessed their applicability in the context of distributed actor-based programming languages. We found that mirrors were a good solution to address the lack of stratification and encapsulation. As a matter of fact, every actor now has its base- and meta-level behaviour separated in distinct objects with their own scope. Additionally, meta-methods can be *exported* to the base-object while their body is still evaluated in the scope of the meta-object, and the programmer can address an actor's meta-behaviour directly through the μ-notation described in section 5.1.2.

However, mirrors do not directly address the concurrency issues typically associated with distributed environments. To deal with them, we opted for a model in which the base-level and the meta-level of every actor are wrapped in the same actor shell. This design differs significantly from meta-architectures found in other actor-based languages, such as ABCL/R [WY88], which typically use different base- and meta-level actors to implement reflection. This single-actor design proved to be a simple and effective solution to tackle the issues of concurrency and synchronization between the base- and meta-level. The reason for this is that the passive base- and meta-objects inherently communicate with one another using synchronous message passing and that this coincides well with the semantics of shifting from the base-level to the meta-level and vice versa, which are also synchronous operations.

Furthermore, mirror-based designs do not directly provide an answer to the ad hoc composition problem. This is an especially important problem in the context of AmbientTalk, which modularizes meta-level behaviour in so-called *language extensions*. Therefore, a mechanism was required to prevent language extensions (also called *language mixins*) that are combined together from accidentally overriding each other's behaviour. To this end, we extended our mirror-based meta-object protocol for AmbientTalk with a high-level language mixin composition mechanism in the form of a domain-specific language. This allows the programmer to represent language extensions in a structured way and to annotate them with information about e.g. which methods they override and which other language extensions they require. This domain-specific language also allows the programmer to specify in a declarative way a set of language extensions to combine together, such that the system automatically determines *if* these extensions can be combined and *how* they can be combined.

7.1.3 Applicability to other languages

Although we primarily used AmbientTalk as a case study in this dissertation, we claim that the contributions of our research transcend AmbientTalk and should be applicable to other distributed actor-based languages as well. Indeed, the single-actor design presented in chapter 5 has many "conceptual" benefits. For example, it naturally supports synchronization when switching between the base-level and the meta-level of an actor. While this reduces concurrency in the system, it is well in line with AmbientTalk's policy that forbids more than one thread from accessing an actor's internal data at the same time. Furthermore, the stratification and encapsulation principles draw a clear line between base- and meta-level operations, protecting each one from accidental changes and overriding from the other. Similarly, the composition mechanism presented in chapter 6 is based on the general principle of annotating possibly conflicting language extensions such that these conflicts can be automatically inferred by the composition mechanism. Hence, although it is most likely that another actor-based reflective language will require a different domain-specific composition language, the general architecture should be reusable.

7.2 Limitations

This section explains some of the technical and conceptual limitations of our proposal.

7.2.1 No infinite reflective tower

As explained in section 5.3.1, our mirror-based meta-object protocol lacks a proper tower of meta-levels, as is often found in other reflective architectures [Mae87, WY88]. This limitation stems from the fact that the existing language constructs we considered are designed to influence the base-level only and do not require higher meta-level operations. However, we have seen that the absence of a tower of meta-objects renders the definition of certain operations, such as *meta-become*, difficult and unnatural. What is more, the lack of an infinite reflective tower makes it impossible for language extensions to be used in the implementation of other language extensions in our design. For example, the due-blocks extension cannot be implemented using, e.g. non-blocking futures, because all meta-meta-behaviour is short-circuited in the language interpreter (see also section 5.3.1).

We state that the question of how to implement an infinite reflective tower in our mirror-based design is a challenging one and certainly deserves some further attention.

7.2.2 Extensible language mixin composition

The language mixin composition mechanism described in chapter 6 is still relatively rudimentary. For instance, it relies on the programmer to specify all the information about exported methods, method replacements and refinements, ... An interesting extension would be to automatically deduce this information from the source code of the mixin itself. However, it is doubtful if this can even be achieved at all, considering the extremely dynamic nature of AmbientTalk.

Perhaps more feasible would be the implementation of an extended requirement system based e.g. on a mixin's public interfaces. For instance, a mixin could simply require any mixin that provides the `when` language construct, instead of explicitly requiring the futures mixin. This would result in less coupling between language mixins, because one would couple mixins based on their interface rather than on their implementation.

Also worthwhile would be to relate our work to existing approaches for meta-behaviour composition, such as found in e.g. MOOSTRAP [MMC95]. Additionally, it would be valuable to examine the alternatives to mixin-based composition in AmbientTalk. Because mixin composition is implemented using inheritance, mixins are composed linearly. This causes several problems, one of them being the fact that a suitable total ordering of mixin applications may sometimes be impossible to find (see section 6.4.1). *Traits* [DNS$^+$06] are a possible way of tackling the limitations of mixin-based composition. In a nutshell, a trait is a (stateless) set of methods, divorced from any object delegation hierarchy. Traits can be composed in any order, and leave the composite entity complete control over the composition. Furthermore, the composite entity can resolve conflicts explicitly, without resorting to linearization [DNS$^+$06].

7.2.3 No strict stratification of the base- and meta-levels

Recall from section 5.3.1 that our mirror-based meta-object protocol makes some small concessions to the stratification principle of mirror-based designs. Indeed, in languages that follow the mirror-based design principles to the letter (e.g. Self), adherence to the stratification principle allows the programmer to dynamically add (remove) reflection support to (from) the language. However, the question of how to support behavioural intercession in such settings remains an open one [BU04]. Nevertheless, because AmbientTalk serves mainly as an exploratory platform for experimenting with new language constructs and because AmbientTalk's meta-level architecture is therefore not meant to be oblivious from the base-level (i.e. the base-level actively makes use of the language constructs applied to it via the meta-level), we judged it acceptable to somewhat loosen the requirements of the stratification principle to allow for more powerful intercession. Nonetheless, by doing so we took care never to break the encapsulation principle in our mirror-based meta-object protocol.

7.3 Pointers for future research

The future work stated in the previous section covers the most "conceptual" limitations of this dissertation. This section proposes some pointers for more "pragmatic" future research in the area of open implementations for ambient-oriented applications.

Recall that we briefly raised some arguments in favour of wholly opening up AmbientTalk's service discovery protocol to the programmer in section 4.1.3. This would, for example, make it possible to define custom comparator methods to compare an actor's list of requirements with other actors' lists of provided services. For instance, such comparators could take into account the state of the actor in their matching process.

Apart from the service discovery protocol, one could also experiment with opening up other aspects of the language, such as the message delivery protocol. Recall from section 3.4 that the default policy of the languages is to guarantee *eventual* delivery of every message. Reifying the transmission protocol could, for example, allow the programmer to use “send once” semantics for sending messages [CDMM05], e.g. when the available bandwidth is low.

A
Default Actor Meta-object Implementation

The AmbientTalk source code of the default actor meta-object in the mirror-based meta-object protocol is presented below.

```
createActorMetaBehaviour :: {

  actorMetaBehaviour:object({

    mailbox::object({
      name: "in";

      cloning.new(aName)::name:=aName;
      add(el):: .add(name, el);
      get(nr)::messages(name)[nr];
      addTable(tbl):: .addAll(name, tbl);
      delete(el):: .delete(name, el);
      length()::size(messages(name));
      init(contents):: .initMailbox(name, contents);

      removeIf(condition(msg)):: {
        newContents: vector.new();
        this().asVector().iterate({
          if(not(condition(el)), newContents.add(el))
        });
        init(newContents.asTable())
      };

      asVector()::{
        tbl: messages(name);
        vector.newWithTable(tbl)
      }

    });
```

```
inbox      : mailbox.new("in");
outbox     : mailbox.new("out");
sentbox    : mailbox.new("sent");
rcvbox     : mailbox.new("rcv");
required   : mailbox.new("required");
provided   : mailbox.new("provided");
joinBox    : mailbox.new("joined");
disjoinBox: mailbox.new("disjoined");

cloning.new()::{
  inbox      := mailbox.new("in");
  outbox     := mailbox.new("out");
  sentbox    := mailbox.new("sent");
  rcvbox     := mailbox.new("rcv");
  required   := mailbox.new("required");
  provided   := mailbox.new("provided");
  joinBox    := mailbox.new("joined");
  disjoinBox:= mailbox.new("disjoined")
};

init():: void;

in(msg)::void;
rcv(msg)::void;
out(msg)::void;
sent(msg)::void;
join()::void;
disjoin()::void;
joined(aResolution)::void;
disjoined(aResolution)::void;

process(message)::{
  message.process(this());
  rcvbox.add(message)
};

createMessage :: { ‘ MOP method to create new messages ‘
  message : object({
    source : void;   target  : void;
    name   : void;   argList : void;

    cloning.new(aSource, aTarget, aName, anArgList)::{
      source:=aSource;   target:=aTarget;
      name:=aName;       argList:=anArgList
    };

    getSource()::source;   setSource(aSource)::source:=aSource;
    getTarget()::target;   setTarget(aTarget)::target:=aTarget;
    getName()::name;       setName(aName)::name:=aName;
    getArgs()::argList;    setArgs(anArgList)::argList:=anArgList;

    process(behaviour)::behaviour.execute(this())
  });
```

```
      message.new
    };

    send(msg)::{
      outbox.add(msg);
      void
    }

  });

  actorMetaBehaviour.new

};

rootActorMetaBehaviour::createActorMetaBehaviour();
defaultMetaMessage:: rootActorMetaBehaviour.createMessage(void, void,
                                                          void, void)
```

B
Source Code of the Mixin Composition Mechanism

The complete source code of the language mixin composition mechanism for AmbientTalk is presented below. It is explained in greater detail in chapter 6.

```
‘############# Simple set #############‘
smallset:: vector.extend({
    add(el):: if(not(contains(el)), .add(el))
});

‘############# Directed graph #############‘

directedGraph:: object({
    nodes: smallmap.new();

    cloning.new(aNodesTable):: map(addNode(el), aNodesTable);

    cloning.clone(aDag):: {
        nodes := smallmap.new();
        aDag.getNodes().iterate({
            addNode(el);
            from: el;
            aDag.edges(from).iterate(addEdge(from, el))
        })
    };

    containsNode(aNode):: nodes.containsKey(aNode);

    addNode(aNode):: if(not(containsNode(aNode)),
                        nodes.put(aNode, smallset.new()));

    getNodes():: nodes.getKeysVector();
```

```
edges(aNode):: nodes.get(aNode);

addEdge(from, to):: {
  addNode(from);
  addNode(to);
  edges(from).add(to)
};

removeEdge(from, to):: edges(from).remove(to);

nEdges():: {
  n: 0;
  nodes.iterate(n := n + value.length());
  n
};

nEdgesTo(aNode):: {
  n: 0;
  iterate(if(edges.contains(aNode), n:= n+1));
  n
};

iterate(it(node, edges)):: nodes.iterate(it(key, value));

print():: iterate({
    display("Node ", node, ": ");
    edges.print()
});

topologicalSort(error(graph)):: {
    g: clone(this());
    nodesWithNoIncomingEdges():: {
        resNodes: vector.newWithVector(g.getNodes());
        g.iterate(edges.iterate(resNodes.remove(el)));
        resNodes
    };
    q: nodesWithNoIncomingEdges();
    result: vector.new();
    n: void;
    adjNodes: void;
    while(or(g.nEdges() > 0, q.length() > 0), {
        if(q.length() = 0, error(g));
        n:= q.delete(1);
        result.add(n);
        ` take a copy because the iterator removes `
        ` elements from the container `
        adjNodes:= vector.newWithVector(g.edges(n));
        adjNodes.iterate({
            g.removeEdge(n, el);
            if(g.nEdgesTo(el) = 0, q.add(el))
        })
    });
    result
    }

});

'############# Composer containing a meta-behaviour and a ############# '
'############# list of mixins already applied to it        ############# '

makeMixinComposer:: {

  composerP:: object({
    behaviour: void;
    appliedMixins: void;
    exportedMethods: void;
    pendingMixins: void;

    cloning.new(aBehaviour):: {
      behaviour  := aBehaviour;
      appliedMixins  := vector.new();
      exportedMethods  := vector.new();
      pendingMixins:= smallset.new()
    };

    isAlreadyApplied(aMixin): appliedMixins.contains(aMixin);
    conflictingExports(aMixin): containsAny(aMixin.getExported(),
                                        exportedMethods);

    applyMixin(aMixin): {
      if(not(isAlreadyApplied(aMixin)),
         if(conflictingExports(aMixin),
            error("Trying to export same method twice. Cannot continue"),
            { behaviour := aMixin.applyTo(behaviour);
              exportedMethods.addTable(aMixin.getExported());
              appliedMixins.add(aMixin) }))
    };

    computeRequired(aMixinsTable): {
      required: directedGraph.new(aMixinsTable);
      ' we add a directed edge A->B for each mixin A that requires B '
      loopRequired(aMixin):: {
        map({ required.addEdge(aMixin, el);
              loopRequired(el) },
            aMixin.getRequired())
      };
      ' Compute requirements for each mixin provided '
      map(loopRequired(el), aMixinsTable);
      required
    };

    computeMixinGraph(aMixinsTable): {
      required: computeRequired(aMixinsTable);
      ' take a copy because the iterator adds elements to the graph '
      allMixins: vector.newWithVector(required.getNodes());
      allMixins.iterate({
          mixinA: el;
          allMixins.iterate({
              mixinB: el;
              ‘ If a mixin A replaces a method replaced or refined by B, ‘
              ‘ we add a directed edge B->A ‘
              if(mixinB.replaces(mixinA),
                 required.addEdge(mixinA, mixinB));
              if(mixinA.replaces(mixinB),
                 required.addEdge(mixinB, mixinA))
          })
      });
      conflictsGraph: void;
      mixinsInOrder: required.topologicalSort(conflictsGraph := graph);
      if(is_void(conflictsGraph),
         { mixinsInOrder := mixinsInOrder.reverse();
           display("Applying mixins in order: ");
           mixinsInOrder.iterate(display(el.getName(), ", "));
           mixinsInOrder },
         ‘ A cycle has occured (A->B, B->A). The mixins that  ‘
         ‘ are still in the graph are the conflicting mixins. ‘
         { display("Conflicts due to mixins: ");
           conflictsGraph.iterate(display(node.getName(), ", "));
           display(eoln);
           error("Conflicts in mixin composition.") })
    };

    ‘ Apply a table of mixins objects ‘
    applyMixins(aMixinsTable): {
      graph: computeMixinGraph(aMixinsTable);
      graph.iterate(applyMixin(el))
    };

    addMixin(aMixin):: pendingMixins.add(aMixin);

    compose():: {
      graph: computeMixinGraph(pendingMixins.asTable());
      graph.iterate(applyMixin(el));
      pendingMixins := smallset.new();
      this()
    };

    isComposer():: true;

    getBehaviour():: behaviour

  });

  composerP.new

};

languageMixin(name,
              exports_clause(exports),
              requires_clause(requires),
              replaces_clause(replaces),
              refines_clause(refines),
              implements_clause(implements(fun(bhv)))):: {

    `############# Structured mixin prototype #############`
    mixinP:: object({
      name: void;
      exported: void;
      required: void;
      replaced: void;
      refined: void;
      implementation: void;

      cloning.new(aName, anExports, aRequired,
                  aReplaces, aRefines, anImplementor):: {
        name := aName;
        exported := anExports;
        required := aRequired;
        replaced := aReplaces;
        refined := aRefines;
        implementation := anImplementor
      };

      getName():: name;
      getRequired():: required;
      requires(aMixin):: containsAll([aMixin], required);
      getMetaMethods():: append(replaced, refined);
      getReplaced():: replaced;
      replaces(aMixin):: containsAny(replaced,
                                     aMixin.getMetaMethods());
      getRefined():: refined;
      refines(aMixin):: containsAny(refined,
                                    aMixin.getMetaMethods());
      getExported():: exported;
      exports(aMethodName):: containsAll([aMethodName], exported);

      applyTo(behaviour):: implementation(behaviour)

    });

    ` use pico reflection to extract the name of the method `
    methodName(method):: method[1][3][1];
    exports_fun@args():: map(methodName(el), args);
    requires_fun@args:: args;
    replaces_fun@args():: map(methodName(el), args);
    refines_fun@args():: map(methodName(el), args);

    mixinP.new(name,
               exports_clause(exports_fun),
               requires_clause(requires_fun),
               replaces_clause(replaces_fun),
               refines_clause(refines_fun),
               { extender(beh):: implements_clause(fun(beh)) })
};
```

```
composer ++ mixin:: {
    c: void;
    if(and(has_slot(composer, "isComposer"),
           composer.isComposer()),
       c := composer,
       ‘ Assume that "composer" is a meta-behaviour passive obj ‘
       c := makeMixinComposer(composer));
    c.addMixin(mixin);
    c
};

compose(composer):: composer.compose().getBehaviour()
```

Bibliography

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[Agh90] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[Arn99] Ken Arnold. The jini architecture: dynamic services in a flexible network. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 157–162, New York, NY, USA, 1999. ACM Press.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, MA, USA, 1985.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[BDG$^+$88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Not.*, 23(SI):1–142, 1988.

[BdR88] J.-P. Briot and J. de Ratuld. Design of a distributed implementation of abcl/i. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 15–17, New York, NY, USA, 1988. ACM Press.

[Bel] Tom Bellwood. Oasis UDDI Specifications TC - Committee Specifications. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.

[BGH00] Richard C. Braley, Ian C. Gifford, and Robert F. Heile. Wireless personal area networks: an overview of the ieee p802.15 working group. *SIGMOBILE Mob. Comput. Commun. Rev.*, 4(1):26–33, 2000.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[Bud01] Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 3rd edition, October 2001.

[CDMM05] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, and Wolfgang De Meuter. A Meta-level Architecture for Ambient-aware Objects. In *First workshop on Object Technology for Ambient Intelligence (ECOOP)*, Glasgow, Scotland, 2005.

[CHDS94] Wim Codenie, Koen De Hondt, Theo D'Hondt, and Patrick Steyaert. Agora: message passing as a foundation for exploring OO language concepts. *SIGPLAN Not.*, 29(12):48–57, 1994.

[CM04] Tom Van Cutsem and Stijn Mostinckx. A prototype-based approach to distributed applications. Master's thesis, Vrije Universiteit Brussel, 2004.

[DB04] Jessie Dedecker and Werner Van Belle. Actors for mobile ad-hoc networks. In *EUC*, pages 482–494, 2004.

[D'Ha] Theo D'Hondt. Interpretation of Computer Programs II. http://prog.vub.ac.be/ tjdhondt/ICP2/HTM.dir/introduction.htm.

[D'Hb] Theo D'Hondt. The Pico Programming Language. http://pico.vub.ac.be/.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: from a new taxonomy to constructive proposals and their validation. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 201–217, New York, NY, USA, 1992. ACM Press.

[DNS$^+$06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[DVM$^+$06] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object- oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer, 2006. To Appear.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[KP94] Gregor Kiczales and Andreas Paepcke. Open implementations and metaobject protocols. Expanded tutorial notes, 1994. At http://db.stanford.edu/˜paepcke/shared-documents/Tutorial.ps.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[MDD04] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Pico: Scheme for mere mortals. In *1st European Lisp and Scheme Workshop*, Oslo, Norway, 2004.

[Meu04] Wolfgang De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks.* PhD thesis, Vrije Universiteit Brussel, 2004.

[Mic] Sun Microsystems. Java platform debugger interface. http://java.sun.com/products/jpda/.

[MMAY95] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 300–315, New York, NY, USA, 1995. ACM Press.

[MMC95] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 316–330, New York, NY, USA, 1995. ACM Press.

[MTM$^+$05] Wolfgang De Meuter, Eric Tanter, Stijn Mostinckx, Tom Van Cutsem, and Jessie Dedecker. Flexible object encapsulation for ambient-oriented programming. In *Proceedings of the Dynamic Language Symposium - Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2005.

[MTS05] M. Miller, E. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. *Synopsium on Thrustworthy Global Computing*, 3705:195–229, 2005.

[Pae93] Andreas Paepcke. User-level language crafting. In *Object-Oriented Programming : the CLOS perspective*, pages 66–99. MIT Press, 1993.

[RHH85] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[SDNB02] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. Technical report, 2002.

[Smi82] B. Smith. Reflection and Semantics in a Procedural Language. Technical Report MIT-TR-272, Massachusetts Institute of Technology. Laboratory for Computer Science, Cambridge, Massachusetts, 1982.

[TK01] R. Tolksdorf and K. Knubben. dself - a distributed self. Technical Report KIT-Report 144, TU Berlin, 2001.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[VA98] Carlos A. Varela and Gul A. Agha. What after java? from objects to actors. *Comput. Netw. ISDN Syst.*, 30(1-7):573–577, 1998.

[Wey78] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. Technical report, Stanford, CA, USA, 1978.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 306–315, New York, NY, USA, 1988. ACM Press.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268, New York, NY, USA, 1986. ACM Press.