Design of a Multi-Level Reflective Architecture for Ambient Actors

Dieter Standaert[1] Éric Tanter[1]⋆ Tom Van Cutsem[2]⋆⋆

[1] DCC, University of Chile
Santiago – Chile
{dstandae,etanter}@dcc.uchile.cl

[2] PROG Lab, Vrije Universiteit Brussel
Brussels – Belgium
tvcutsem@vub.ac.be

Abstract. Ambient-Oriented Programming (AmOP) languages are especially designed for software development for pervasive and ambient computing. In this context, reflective abilities are highly desired to be able to create adaptive software. In this paper, we propose a multi-level reflective architecture for ambient actors, instantiated in the AmOP language AmbientTalk. Our architecture is structured according to different levels of abstraction, distinguishing between the metalevel for regular objects, active objects and objects representing aspects of the virtual machine (VM) itself. The architecture adopts at its core the concept of mirrors and mirror methods to safeguard object encapsulation even in the presence of powerful reflective facilities, such as access to the VM.

1 Introduction

In this paper, we approach pervasive computing and ambient intelligence from a software engineering point of view and focus on a metalevel architecture for so-called *ambient-oriented programming* (AmOP) languages [5]. AmOP is a novel paradigm of computing that advocates distributed programming languages that are explicitly geared towards creating software for pervasive ad hoc networks populated by mobile and/or embedded devices.

It is a given fact that distributed programming is a difficult task. Pervasive and ambient computing further complicate matters as mobile networks feature an open network topology, where devices may join and leave at any point in time, where reliance on central servers is usually impractical and where the connection between two communicating devices is often volatile due to the limited wireless communication range. These phenomena are unavoidable consequences of the hardware and *cannot* be dealt with automatically by a language or library. If no support to deal with the above phenomena is offered by a programming language, a software developer will be forced to deal with them in an ad hoc manner in each application, over and over again.

This paper describes a multi-level reflective architecture to address the above problem. Rather than requiring the developer to manually deal with the difficult

⋆ É. Tanter is financed by the Milenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

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

issues engendered by the ambient hardware at the base level, where this code would be severely tangled with and scattered throughout the functional code, we want to offer the programmer a means to express a solution for the issues in a generic manner, at the metalevel. We structure this metalevel according to different levels of abstraction, which gives rise to what is known as a *multi-model reflection framework* [8]. This structure is simply derived from the fact that not all distribution-related issues are expressible at the same level of abstraction, as will be explained later on.

The paper is organized as follows. In Section 2, we describe the concrete ambient-oriented programming language AmbientTalk [5] in which we instantiate our architecture. Furthermore, we describe *mirror methods* [9], which form the basic language construct to shift from base to metalevel in our architecture and which avoids a number of security issues related to reflection by sticking to the *extreme encapsulation* principle [3]. Section 3 describes the proposed multi-level reflective architecture, identifying different *layers* of abstraction at the metalevel and, for each of these layers, identifying the useful aspects of the language to reify. The conclusion summarizes the salient points of the architecture.

2 Motivation and Related Work

Because our work on ambient actors is realized in the programming language AmbientTalk, we first briefly discuss the major characteristics of this AmOP language [5]. We then focus on two trends in sound design principles for metalevel architectures, which we aim at combining in our proposal: first, we survey related work on how multi-model reflection frameworks (MMRF) enhance the structuring of metafacilities; second, we discuss mirror methods, a language construct that preserves the extreme encapsulation of objects in the presence of reflection.

2.1 AmbientTalk

AmbientTalk [5] is a distributed object-oriented programming language with features that explicitly aid in the construction of software to be deployed on mobile (ad hoc) networks. It has built-in provisions for discovering remote software entities without relying on centralised name servers and for asynchronous communication. The AmbientTalk language discriminates between *passive* and *active* objects. Passive objects are regular objects without any concurrency or distribution provisions, which are used to model typical object-oriented applications. AmbientTalk's passive objects are not instantiated from classes; rather, AmbientTalk has a prototype-based object model where objects are created ex-nihilo or by cloning or extending other objects. This makes objects entirely self-sufficient: they do not depend on a separate class description.

AmbientTalk's active objects are heavily based on *actors* [1], but unlike pure actors which are functional, AmbientTalk actors may be *stateful*. More precisely, AmbientTalk follows the *ambient actor* model [4]. In this model, the behavior

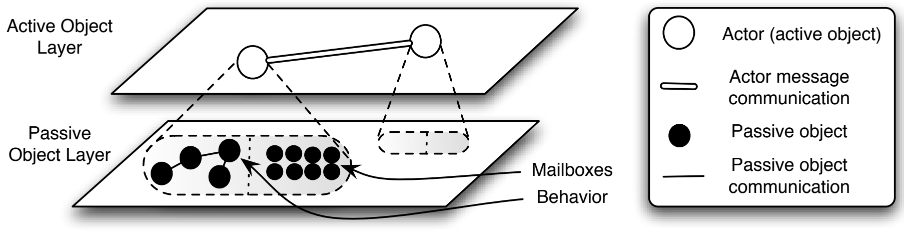

Fig. 1. The AmbientTalk double-layered object model: an active object is implemented by a behavior passive object (and aggregates), as well as 8 mailboxes.

and state of an actor is implemented by passive objects. Every actor has one special passive object which is called its *behavior*. The methods which this passive object implements are used to process the incoming messages of the actor. All of the other passive objects to which the behavior object may point are never shared with other actors. Passing passive objects in messages to other actors results in a pass-by-copy semantics, enforcing the *containment* principle of ambient actors, which shields passive objects from concurrent accesses. Figure 1 illustrates AmbientTalk's double-layered object model.

In addition to its behavior, an ambient actor is characterized by a number of *mailboxes*. Some mailboxes are used for actor communications, while others are used for service discovery. First, four mailboxes are used to store the messages an actor has received, processed, sent and delivered. These mailboxes are called the `inbox`, the `rcvbox`, the `outbox` and the `sentbox`. An actor encapsulates *exactly one* thread of execution, which is an eternal event loop, taking a message from its actor's `inbox`, processing it and putting it in its `rcvbox`. Because an actor has but one thread, race conditions within an actor cannot occur. Communication between two actors is always *asynchronous*. When an actor `a` sends a message to another actor `b`, it puts the message in its `outbox`, addressed to `b`. The actor system (*i.e.* the virtual machine) is responsible for delivering this message by putting it in `b`'s `inbox`. When the message has been delivered, the actor system moves the message from `a`'s `outbox` to its `sentbox`. The beneficial properties of asynchronous communication in mobile networks are described elsewhere [5, 7].

AmbientTalk actors can discover one another via AmbientTalk's *service discovery* mechanism. This mechanism consists of another set of mailboxes. An actor may advertise its services on the network by adding *patterns* to its `providedbox`. This mailbox describes what services an actor *provides*. Conversely, an actor may indicate to the discovery mechanism that it *requires* a certain service by adding a pattern to its `requiredbox`. When two devices running an AmbientTalk virtual machine encounter one another, they exchange these required and provided patterns. When a match is detected, the actor requiring the service is notified. Notification happens via a dedicated mailbox known as the `joinedbox`: when a provider actor is found, a reference to this actor is added to the requiring actor's `joinedbox`. When the provider actor leaves the network, the reference is moved from the requiring actor's `joinedbox` to its `disjoinedbox`. In short, these four mailboxes allow an actor to discover other actors in its ambient.

The current implementation of AmbientTalk uses the dual-layer object model described above, but its reflection architecture permits only the introspection of the mailboxes of an actor. We now turn to several metalevel architectures that incarnate sound design principles, and which we aim at combining in a comprehensive reflective architecture for AmbientTalk and its dual-layer object model.

2.2 Multi-Model Reflection Frameworks

Okamura et al [8] describe a *multi-model* reflection framework (MMRF) for a distributed programming system known as AL-1/D. The motivation behind the MMRF is to properly structure the metalevel so that several *views* on base-level objects are provided. In other words, the MMRF does not associate a single metaobject with each base object, but rather a set of metaobjects, each providing a representation of the base-level object at a different level of abstraction. For instance, AL-1/D provides the following metamodels:

– An **operation model** describing message sends and the behavior of objects.
– A **resource model** representing an object at the virtual machine level.
– A **statistics model** giving information of the object's state and resource occupation in the environment.
– A **distributed environment model**, reifying an object's location, the semantics of remote method invocation and other objects in its environment.
– A **migration model** describing how objects may migrate to remote hosts.
– A **system model** describing the semantics of the above models themselves.

As noted by Okamura *et al.* [8], the advantage of such an approach is that the metalevel programmer may choose the right level of abstraction to fit his needs. For example, for some metaprograms viewing an object as a structured collection of method and field slots is a convenient representation, while other, more low-level metaprograms (such as a garbage collector) may want to view objects as segments in memory. In a context as complex as distributed computing, even more pervasive and ambient computing, the design principle of the MMRF is highly valuable to simplify the metalevel interface according to semantical views.

The application of a MMRF to ambient actors in AmbientTalk has been explored by one of the authors in previous work [10]. The previously-described metalevel architecture is structured according to a distinction between structural and behavioral reflection and consists of three layers:

1. The **base level** consisting of actors, their mailboxes and the messages they send and receive.
2. A **structural metalevel** providing a reification of messages and an actor's behavior, mailboxes, required and provided services and its identity.

```
makePoint(aX,aY):: object({
   x:aX; y:aY;
   moveTo(nX,nY)::{ x:=nX; y:=nY; }
   cloning.copy()::{ this; }
   mirror.invoker()::{
      methods()::{ meta.methods(); }}
});
```

Fig. 2. Prototype point object featuring a cloning method and a mirror method.

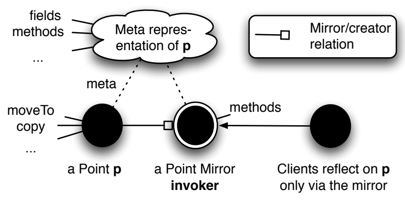
fields
methods
...
Meta representation of **p**
Mirror/creator relation
meta
moveTo
copy
...
methods
a Point **p**
a Point Mirror **invoker**
Clients reflect on **p** only via the mirror

Fig. 3. Mirror on a (passive) point object.

3. A **behavioral metalevel** defining an actor's message delivery protocol, service discovery protocol and a distributed garbage collection protocol.

The present work builds upon that metalevel architecture. However, rather than structuring the metalevel according to the structural/behavioral distinction, we propose a different structuring based on levels of abstraction. The structural and behavioral parts of a reified concept are now bundled together. Furthermore, the architecture described in [10] does not describe a language construct to ensure that reflection does not break encapsulation. This has been addressed by mirror methods, explained below.

2.3 Mirror methods

A fundamental principle in the context of open networks is that of *extreme encapsulation* [3]: an object should be subject to message passing only, retaining control over all operations done over it. This precludes the provision of omnipotent cloning or reflection operators, which can act without the object's consent.

AmbientTalk already features *cloning methods*, *i.e.* methods that when invoked, result in the creation of a clone of the receiver, and the evaluation of the method body in the scope of the clone (Fig 2, method `copy`). The advantage of a cloning method over a clone operation is that the method body can be used to initialize the clone without any need to expose potential encapsulation breaching accessors to allow state modification from the outside. This concept has been extended to reflection, by relying on the notion of *mirror methods* [9]. First of all, *mirrors* have been proposed as a structuring mechanism for metalevel facilities: instead of reflecting on an object directly, metalevel facilities are exposed by special objects called mirrors [2]. Relying on mirrors serves a number of design principles, such as encapsulation (they separate the implementation of reflective facilities from their interface), stratification and ontological correspondence. We refer the interested reader to [2] for details.

A mirror method is a method whose evaluation results in the creation of a mirror object providing metalevel facilities on the receiver. The precise set of metalevel facilities provided by a mirror is controlled by the receiver itself, since the body of the mirror method determines the expressive power of the mirror, *i.e.* what it has access to. In [9], a reflective API based on mirror methods for ChitChat – the precursor of AmbientTalk – is presented. It relies on the introduction of a new pseudo-variable, `meta`, which gives access to the metalevel representation of an object. Any object has a reference to its metalevel representation, has full access over it, and can hand out a selected set of capabilities to the outside world via mirrors. A mirror object shares the `meta` of its creator. Fig. 2 shows a mirror method `invoker` on points, formulated in AmbientTalk: when invoked, the `invoker` mirror method returns a mirror on the receiver, which has one method, `methods`. Hence, if `p` is a point, `p.invoker().methods()` returns the table of all the methods of `p`. One can then use this table to introspect and invoke methods on `p` reflectively (Fig. 3).

The reflective API presented in [9] covers both structural (access to methods and fields) and behavioral (listeners on execution events in the object) aspects. Furthermore, a number of facilities are provided to control access rights given over reifications. For instance, it is possible to hand out a mirror on a method of an object while ensuring that only the signature of the method can be introspected, but the method cannot be invoked reflectively.

The formulation of mirror methods in [9] is only in the context of passive objects (mirrors are passive objects, mirror methods are declared in passive objects), and hence needs to be extended to be applicable for ambient actors as provided by AmbientTalk.

2.4 Motivation

This work aims at designing a comprehensive reflective architecture for ambient actors in AmbientTalk, by sticking to the fundamental design principles of mirror methods and multi-model reflection. To reach this objective, we extend the concept of mirrors and mirror methods to fit the passive and active object model of AmbientTalk. We can then expose reflection on the different metalevels we consider: reflection on passive objects, on active objects, and on the VM. At each level, a comprehensive set of reflective capabilities is provided.

3 Multi-level Reflective Architecture for Ambient Actors

The proposed reflective architecture for ambient actors as provided by AmbientTalk is illustrated in Fig. 4. It distinguishes three layers of abstraction. The first two layers correspond to the double-layered object model of AmbientTalk, distinguishing between reflecting upon passive or active objects. The third layer is the *virtual machine* layer, which allows a metaprogrammer to reflect upon the low-level machinery upon which the other layers are built. More precisely:

– The metaobject protocol for the **passive object layer** is based on mirror methods as explained in Sect. 2.3.
– At the **active object layer**, we provide an adaptation of mirrors and mirror methods in order to give metalevel facilities on actors respecting the encapsulation properties of mirror methods, but taking into account the issues

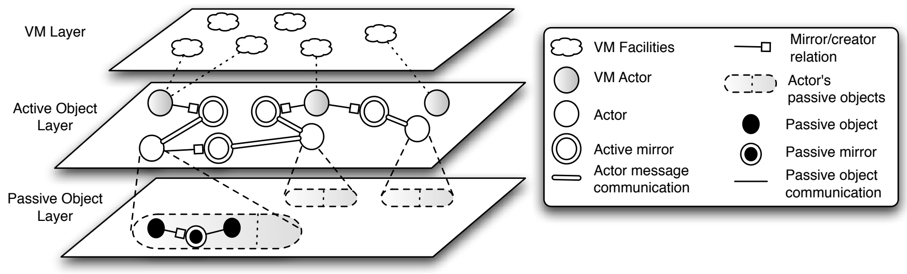

Fig. 4. Overview of the Multi-level Reflective Architecture for Ambient Actors.

proper to actors: concurrent activities, message-based communication, and no sharing of passive objects between actors to avoid race conditions. We introduce the notion of *active mirrors*, which are themselves actors.

– To include reflection on the virtual machine in our language, the facilities of the virtual machine at the **VM layer** are propagated to the level of actors. Actors encapsulating such VM facilities are called *VM actors*: A VM actor provides a set of mirror methods that give controlled and tailored access to the VM facilities it represents, depending for instance on the client requesting access to such facilities (*e.g.* the central VM actor in Fig. 4).

Compared to the current AmbientTalk architecture, our proposal includes mirrors at both object levels (passive and active) in our to respect the extreme encapsulation principle, and also includes specific entities for reflecting upon the VM, a facility that AmbientTalk does not currently provide. Note that meta-level programming is done in AmbientTalk itself, hence explaining why mirrors (either passive or active) live among application objects. The same reason justifies the presence of VM actors at the active object layer (the implementation of AmbientTalk is not metacircular, the VM being written in Java).

3.1 The Passive Object Layer

The metalevel of the passive object layer is accessible via mirror methods as discussed in Sect. 2.3. Mirrors may expose both structural and behavioral information about a passive object. For space reasons we do not recapitulate all the structural and behavioral aspects exposed by mirrors, instead we refer to [9].

A mirror is itself a passive object, which means that it can only be accessed by the actor which owns its base-level object. If a passive mirror object is handed out to another actor, it will be copied along with its base-level object. The receiving actor can hence only reflect on a copy of a base-level object via a copy of its mirror. This decision preserves the containment principle of actors, even at the meta-level.

3.2 The Active Object Layer

The active object layer is populated by actors that communicate via asynchronous message passing, as opposed to synchronous method invocation occurring at the passive object layer. The meta-level is structured according to the same ontology, consisting of *active mirrors*, which are actors describing other actors (this design is similar to the use of active metaobjects in ABCL/R [12]). Communication with active mirrors is realized via asynchronous message passing, which is the only communication abstraction available at this level.

To preserve extreme encapsulation, active mirrors can only be accessed by invoking active mirror methods on base-level actors, analogous to the mirror methods in the passive object layer. This allows a base-level actor to selectively hand out different kinds of active mirrors on itself, offering more or less reflective power depending on the client requesting the active mirror. The facilities that can be handed out in an active mirror describe the core concepts of ambient actors (Sect. 2.1): behavior ("facade" passive object), mailboxes (four mailboxes to regulate message sending, four mailboxes to regulate service discovery) and one thread used to process incoming messages. More precisely:

– **behavior**: an active mirror can offer structural information about the actor based on its passive behavior object. This allows metalevel code to *e.g.* list all messages an actor understands and to send such messages reflectively via active mirrors. An example is given below.
– **mailboxes**: an active mirror can reify the eight mailboxes of an actor. This allows metalevel code to introspect the mailboxes to *e.g.* discover what patterns an actor requires or provides. Furthermore, the active mirror offers a means to replace a mailbox object with an alternative object with a different implementation. For example, one may substitute the default `inbox` mailbox, which is implemented as a FIFO queue, by a *priority* queue such that messages may be processed according to a certain priority. Behavioral reflection at the actor level is achieved by registering observers with mailboxes via the active mirror, corresponding to how behavioral reflection is done with passive mirrors [9]. For instance, a mailbox observer can be triggered each time messages are added to or removed from a mailbox.
– **thread**: reification of the thread of an actor allows for extensive debugging, *e.g.* forcing step-by-step execution of the thread. A reification of the thread could also be used to divert from the standard serialized message processing scheme of base-level actors. For example, an actor's base-level methods may be annotated as read-only and based on this information, meta-level code may fork an actor's thread to process these methods truly in parallel as long as no mutator method is running. Such language customizations are useful for actors in *e.g.* concurrency libraries.

Messages exchanged by actors are passive objects containing the *sender* actor, the *receiver* actor, a *selector* and a table of *arguments*. Messages are reified at the active object layer as mirrors on such passive objects. The mirrors may additionally contain metadata or annotations that can be added and inspected by metalevel code (*e.g.* a message priority level to be taken into account by a priority-based implementation of the `inbox`).

Example. Consider the following domotics actor which represents the interface to a domotics system. The actor provides a number of methods to control the pervasive hardware in a futuristic house:

```
domoticsController :: actor(object({
   openGarage() :: { ... }
   toggleLights(room, brightness) :: { ... }
   ...
   amirror.controls():: {
      messages()::{ metaActor#messages(); }
      receiveMessage(messageMirror) :: {
        metaActor#receiveMessage(messageMirror) }}
}));
```

Asynchronous message sending in AmbientTalk is denoted by a `#` token. An actor may specify a mirror method returning an active mirror using the `amirror` keyword. Within such methods, an actor may refer to its metalevel representation using the `metaActor` keyword. The introduced distinction between `meta` and `metaActor` is exactly the same as the distinction between `this` and `thisActor` in AmbientTalk: the former refers to the (meta of the) passive object itself, while the latter refers to the (meta of the) actor in which the passive object is contained.

The `controls` mirror method allows client actors to introspect all of the messages the controller actor understands, such that they may discover which domotics-related operations it can perform. This allows *e.g.* a user interface actor on a resident's PDA to discover such functionality dynamically and to invoke the controls reflectively (assuming it acquired a mirror on the `toggleLights` message):

```
toggleLightsMsgMirror.setArguments([ "kitchen", 0.8 ]);
aController#controls()#receiveMessage(toggleLightsMsgMirror);
```

3.3 The Virtual Machine and its meta-facilities

The virtual machine of AmbientTalk allows actors to abstract from such hardware details as the type of machine on which they are running and the kind of wireless networking protocol they use for remote communication. It has been noted that in mobile networks, applications may want access to this information because there is much more heterogeneity in the type of devices, operating systems and network protocols used in such networks [6].

Extensible Virtual Machine. The heterogeneity encountered when dealing with mobile networks furthermore requires an ambient-oriented virtual machine

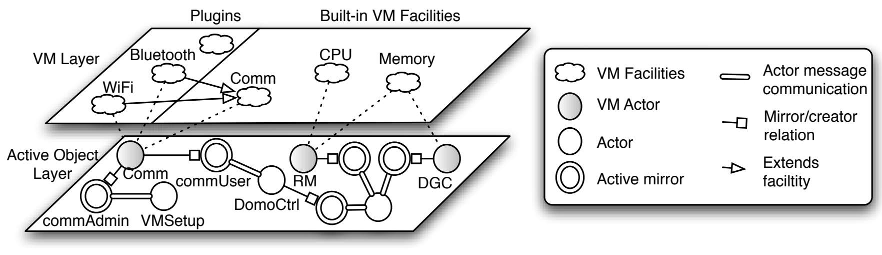

Fig. 5. Exposing VM facilities with VM actors residing at the Active Object Layer.

to be *extensible*. In our approach, the virtual machine supports extensibility in the form of *plugins*. The core functionality of the virtual machine (such as the message delivery protocol or the service discovery protocol) is customizable and extensible via such plugins. A plugin is *not* reflective code, but rather an extension of the VM written in the implementation language of the VM (Java in the case of the AmbientTalk VM). Plugins must be dynamically-loadable so that a choice between different plugins may be postponed until runtime, without requiring all possible plugins to be statically compiled into the VM.

Fig. 5 shows our architecture for exposing metalevel facilities. Note that we discriminate *built-in* facilities and *plugin*-provided facilities. For instance, a communication protocol facility is built in the VM, while the WiFi and Bluetooth extensions are provided via plugins.

VM Actors. The VM should reify its facilities via an interface which allows actors to examine and toggle between available plugins or which allows them to load specific plugins themselves. VM facilities are reified as special actors known as *VM actors* (Fig. 5). Of course, not all actors in the ambient environment should have permission to access all these metafacilities so a means of imposing access control is necessary. This is again achieved via mirror methods. A VM actor does not offer access to the VM services directly, but rather exposes a set of mirror methods with differing access rights depending on what actor invokes the method. These mirror methods return mirror actors which offer the proper methods to access (and modify) VM-specific information.

When extending the VM with plugins, the plugin provider must also provide the corresponding VM actor(s) for base-level actors to interact with the new facilities. This can be done either be defining new VM actors, or by extending existing VM actors. In Fig. 5, the `Comm` VM actor combines the default communication facilities of the VM with the Bluetooth and WiFi extensions. We show an example using this VM actor below.

Typical facilities. The following is an enumeration of VM-specific information which we deem interesting to make available to base-level actors, based on ideas presented in [8, 10], as well as our own findings:

- **Hardware/Software profiling (HSP)**: this service is a reification of all of the facilities a VM has to its disposal, be they hardware facilities (such as a WiFi and Bluetooth card) or software facilities (such as an input or output stream for communication with the user or other programs). An actor may introspect the profile to know which facilities, plugins, etc. are available on the virtual machine. For example, an agent can remotely introspect a VM to ensure it fits a required profile before the agent migrates to it.
- **Network profiling (NP)**: a reification of the identity, position, etc. of the VM in the network.
- **Communication layer (Comm)**: this service is more concerned with the connection type (IPX, TCP/IP, ...) as well as the management of different network protocols (Bluetooth, WiFi, Infrared, etc.).
- **Distributed Garbage Collection (DGC)**: this is a facility to offer actors control over the memory reclamation of actors. To support remote communication, the VM stores special references to local actors which are remotely addressable, such that these actors will not be garbage collected locally. Unfortunately, this precludes an actor from ever being garbage collected at all. Leasing strategies such as employed by the JINI networking technology [11] can be implemented at the metalevel via such a DGC service. A leased reference to an actor must be explicitly renewed in time by remote actors. If it is not renewed before time, the special VM reference to the actor can be cleared via the DGC VM actor such that the local actor can be garbage collected if it is no longer used locally.
- **Resource management (RM)**: a service to offer actors control over the amount of memory and CPU time used. This also includes the scheduling of actor computation. A metalevel programmer may want to assign different priorities to different actors.
- **Discovery protocol (DP)**: a service which controls how services are published on the network. For example, in order to discover other AmbientTalk virtual machines, the VM periodically broadcasts a discovery signal. Actors may want to control the frequency of broadcasting to *e.g.* save battery power. Another aspect of this protocol is how required and provided patterns are encoded and how they are matched. Allowing actors to specify a custom matching or encoding protocol for such patterns allows them to *e.g.* resolve small discrepancies between different versions of the same service interface, or to use Semantic Web ontologies standards such as RDF.
- **Message delivery system (MDS)**: The message delivery system of a virtual machine handles the message transmission between actors. This system is primarily important for being able to change the quality of service (QoS) parameters of a message. Is it guaranteed to be delivered? Is it guaranteed to be delivered at most once, in sequence, . . . ?

Example. Consider a domotics controller actor `DomoCtrl` with the following requirements *w.r.t.* communication protocols: if Bluetooth is available, it is prefered; otherwise, encrypted WiFi should be used if it is available; if not, the default protocol is used. First of all, this requires `DomoCtrl` to be able to *introspect* the available communication protocols, and possibly to *change* the one used for its own communications.

```
Comm :: actor(object({
  amirror.getCommUser():{/* getPtcls(), getMyPtcl(), setMyPtcl(p) */ }
  amirror.getCommAdmin():{/* CommUser + getDefaultPtcl(), setDefaultPtcl(p) */ }
  getCommServices()::{
    if(isAdmin(sender), this.getCommAdmin(), this.getCommUser());}
}))
```

Above is the skeleton of the code for the `Comm` VM actor, which reifies at the actor level the communication layer of the VM (Fig. 5). In order to distinguish between normal users and administrators, its `getCommServices` method checks if the caller (obtained with the keywork `sender`) has administrator privileges or not. If not (*e.g.* `DomoCtrl`), it returns the mirror produced by the mirror method `getCommUser`, which contains only actor-specific services; if yes (*e.g.* a `VMSetup` actor), then the returned mirror includes a more extensive set of services, including the possibility to change the default protocol used for all actors in the VM. This illustrates how reflective access to the VM can be customized via mirrors on VM actors (see Fig. 5).

Now, the adaptive `DomoCtrl` can make use of the `Comm` VM actor as follows in order to properly set its communication protocol during initialization:

```
DomoCtrl :: actor(object({
 init():{
  cs: Comm#getCommServices();
  ptcls: cs#getPtcls();
  if(contains(ptcls, "Bluetooth"), cs#setMyPtcl("Bluetooth"),
     if(contains(ptcls, "Encrypted-WiFi"),
        cs#setMyService("Encrypted-WiFi")))
 }...}));
```

First, the list of communication services is obtained. Since `DomoCtrl` is not an administrative actor, the `commUser` active mirror is returned (bound to `cs`). Then, the actor checks if Bluetooth is available, and if so, sets *its* communication protocol to Bluetooth. Else, if encrypted WiFi is available, it is set, otherwise no change is done so the global default protocol will be used.

4 Conclusion

We have proposed a multi-level reflective architecture for ambient actors and its instantiation in the AmOP language AmbientTalk. Our architecture combines *(a)* the engineering benefits of *multi-model reflection* [8] by structuring meta-level facilities according to different levels of abstraction (passive objects, active objects, virtual machine), *(b)* the extreme encapsulation properties of *mirror methods* [9] by ensuring that objects that are reflected upon can themselves restrict access to their meta-level facilities depending on the client, and this at all levels, and *(c)* the power offered by an *extensible virtual machine* in which facilities are made accessible to actors so that they can customize their execution environment, as well as adapting their own behavior according to it. Compared to the actual MOP provided in AmbientTalk [5], our proposal goes further with respect to the considered facilities (the AmbientTalk MOP is restricted to the actor layer), and respects the extreme encapsulationp principle thanks to its systematic use of mirror methods.

The implementation of the proposed architecture is currently on-going. Future work includes addressing performance considerations, refining our design of the virtual machine layer and actors, and performing concrete experiments.

References

[1] G. Agha. *ACTORS: a model of concurrent computation in distributed systems.* The MIT Press: Cambridge, MA, 1986.

[2] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 331–344, Vancouver, British Columbia, Canada, Oct. 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[3] W. De Meuter, É. Tanter, S. Mostinckx, T. Van Cutsem, and J. Dedecker. Flexible object encapsulation for ambient-oriented programming. In *ACM Dynamic Language Symposium (DLS 2005)*, San Diego, CA, USA, Oct. 2005.

[4] J. Dedecker and W. Van Belle. Actors for mobile ad-hoc networks. In L. Yang, M. Guo, J. Gao, and N. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer-Verlag, Aug. 2004.

[5] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. In D. Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP 2006)*, Lecture Notes in Computer Science, Nantes, France, July 2006. Springer-Verlag. To Appear.

[6] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[7] C. Mascolo, L. Capra, and W. Emmerich. Mobile computing middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.

[8] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on Reflection and Meta-level Architectures*, pages 36–47, Tokyo, Japan, Nov. 1992.

[9] É. Tanter. Mirror methods — reconciling reflection and extreme encapsulation. In *ECOOP Workshop on Object Technology for Ambient Intelligence*, July 2005.

[10] T. Van Cutsem, J. Dedecker, S. Mostinckx, and W. De Meuter. A meta-level architecture for ambient-aware objects. In *ECOOP Workshop on Object Technology for Ambient Intelligence*, July 2005.

[11] J. Waldo. Constructing ad hoc networks. In *IEEE International Symposium on Network Computing and Applications (NCA'01)*, 2001.

[12] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In N. Meyrowitz, editor, *Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 88)*, pages 306–315, San Diego, California, USA, Sept. 1988. ACM Press. ACM SIGPLAN Notices, 23(11).