# Ambient-Oriented Programming in AmbientTalk

Jessie Dedecker[⋆], Tom Van Cutsem[⋆], Stijn Mostinckx[⋆⋆],
Theo D'Hondt, and Wolfgang De Meuter
`jededeck | tvcutsem | smostinc | tjdhondt | wdmeuter@vub.ac.be`

Programming Technology Laboratory
Vrije Universiteit Brussel, Belgium

**Abstract.** A new field in distributed computing, called Ambient Intelligence, has emerged as a consequence of the increasing availability of wireless devices and the mobile networks they induce. Developing software for mobile networks is extremely hard in conventional programming languages because the network is dynamically demarcated. This leads us to postulate a suite of characteristics of future *Ambient-Oriented Programming* languages. A simple reflective programming language, called AmbientTalk, that meets the characteristics is presented. It is validated by implementing a collection of high level language features that are used in the implementation of an ambient messenger application .

## 1 Introduction

Software development for mobile devices is given a new impetus with the advent of *mobile networks*. Mobile networks surround a mobile device equipped with wireless technology and are demarcated dynamically as users move about. Mobile networks turn isolated applications into cooperative ones that interact with their environment. This vision of ubiquitous computing, originally described by Weiser [38], has recently been termed *Ambient Intelligence* (AmI for short) by the European Council's IST Advisory Group [12].

Mobile networks that surround a device have several properties that distinguish them from other types of networks. The most important ones are that connections are volatile (because the communication range of wireless technology is limited) and that the network is open (because devices can appear and disappear unheraldedly). This puts extra burden on software developers. Although system software and networking libraries providing uniform interfaces to the wireless technologies (such as JXTA and M2MI [21]) have matured, developing application software for mobile networks still remains difficult. The main reason for this is that contemporary programming languages lack abstractions to deal with the mobile hardware characteristics. For instance, in traditional programming languages failing remote communication is usually dealt with using a classic exception handling mechanism. This results in application code

⋆ Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)
⋆⋆ Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

polluted with exception handling code because failures are the rule rather than the exception in mobile networks. Observations like this justify the need for a new *Ambient-Oriented Programming* paradigm (AmOP for short) that consists of programming languages that explicitly incorporate potential network failures in the very heart of their basic computational steps.

The goal of our research is threefold:

- First, we want to gain insight in the structure of AmOP applications.
- Second, we want to come up with AmOP language features that give programmers expressive abstractions that allow them to deal with the characteristics of the mobile networks.
- Third, we want to distill the fundamental semantic building blocks that are at the scientific heart of AmOP language features in the same way that current continuations are at the heart of control flow instructions and environments are the essence of scoping mechanisms.

As very little experience exists in writing AmOP applications, it is hard to come up with AmOP language features based on software engineering requirements. Therefore, our research departs from the hardware phenomena that fundamentally distinguish mobile from stationary networks. These phenomena are listed in section 2.1 and form the basis from which we distill a number of fundamental programming language characteristics that define the AmOP paradigm. These characteristics are the topic of section 3. A concrete scion of the AmOP paradigm — called AmbientTalk — is presented starting from section 4. AmbientTalk's design is directly based on our analysis of the hardware phenomena and features a number of fundamental semantic building blocks designed to deal with these hardware phenomena. Since AmbientTalk was conceived as a reflectively extensible language kernel, the semantic buidling blocks turn AmbientTalk into a language laboratory that allows us to investigate the language features that populate the AmOP paradigm. Section 5 validates this by realising three language features that facilitate high-level collaboration of objects running on devices connected by a mobile network. The language features are used in a concrete experiment we conducted, namely the implementation of an ambient peer-to-peer instant messaging application that was deployed on smart phones.

## 2 Motivation

The hardware properties of the devices constituting a mobile network engender a number of phenomena that have to be dealt with by distributed programming languages and/or middleware. We summarize these hardware phenomena below and describe how existing programming languages and middleware fail to deal with them. These shortcomings form the main motivation for our work.

### 2.1 Hardware Phenomena

With the current state of commercial technology, mobile devices are often characterised as having scarcer resources (such as lower CPU speed, less memory and limited battery life) than traditional hardware. However, in the last couple of years, mobile devices and full-fledged computers like laptops are blending more and more. That is why we do not consider such restrictions as fundamental as the following phenomena which are inherent to mobile networks:

**Connection Volatility.** Two devices that perform a meaningful task together cannot assume a stable connection. The limited communication range of the wireless technology combined with the fact that users can move out of range can result in broken connections at any point in time. However, upon re-establishing a broken connection, users typically expect the task to resume. In other words, they expect the task to be performed in the presence of a volatile connection.

**Ambient Resources.** If a user moves with his mobile device, remote resources become dynamically (un)available in the environment because the availability of a resource may depend on the location of the device. This is in contrast with stationary networks in which references to remote resources are obtained based on the explicit knowledge of the availability of the resource. In the context of mobile networks, the resources are said to be ambient.

**Autonomy.** Most distributed applications today are developed using the client-server approach. The server often plays the role of a "higher authority" which coordinates interactions between the clients. In mobile networks a connection to such a "higher authority" is not always available. Every device should be able to act as an autonomous computing unit.

**Natural Concurrency.** In theory, distribution and concurrency are two different phenomena. For instance in a client-server setup, a client device might explicitly wait for the results of a request to a serving device. But since waiting undermines the autonomy of a device, we conclude that concurrency is a natural phenomenon in software running on mobile networks.

### 2.2 Distributed Languages

To the best of our knowledge no distributed language has been designed that deals with all the characteristics of mobile hardware just described. Languages like Emerald [19] and Obliq [6] are based on synchronous communication which is irreconcilable with the autonomy and the connection volatility characteristics. Languages like ABCL/f [31] and Argus [23] that promote a scheme based on futures [14] partially solve this but their objects block when accessing unresolved futures. Other languages based on the actor model, such as Janus [20], Salsa [35] and E [27] use pure asynchronous communication. However, these languages offer no support to discover ambient resources or to coordinate interactions among autonomous computing units in the face of volatile connections.

### 2.3 Distributed Middleware

An alternative to distributed languages is middleware. Over the past few years a lot of research has been invested in middleware for mobile networks [25]. It can be categorised in several groups.

**RPC-based Middleware** like Alice [13] and DOLMEN [29] are attempts that focus mainly on making ORBs suitable for lightweight devices and on improving the resilience of the CORBA IIOP protocol against volatile connections. Others deal with such connections by supporting temporary queuing of RPCs [18] or by rebinding [30]. However, these approaches remain variations of synchronous communication and are thus irreconcilable with the autonomy and connection volatility phenomena.

**Data Sharing-oriented Middleware** tries to maximize the autonomy of temporarily disconnected mobile devices using weak replica management (cf. Bayou [32], Rover [18] and XMiddle [40]). However, since replicas are not always synchronisable upon reconnection, potential conflicts must be resolved at the application level. In spite of the fact that these approaches foster fruitful ideas to deal with the autonomy characteristic, to the best of our knowledge, they do not address the ambient resources phenomenon.

**Publish-subscribe Middleware** adapts the publish-subscribe paradigm [10] to cope with the characteristics of mobile computing [7, 5]. Such middleware allows asynchronous communication, but has the disadvantage of requiring manual callbacks to handle communication results, which severely clutters object-oriented code.

**Tuple Space based Middleware** [28, 24] for mobile computing has been proposed more recently. A tuple space [11] is a shared data structure in which processes can asynchronously publish and query tuples. Most research on tuple spaces for mobile computing attempts to distribute a tuple space over a set of devices. Tuple spaces are an interesting communication paradigm for mobile computing. Unfortunately, they do not integrate well with the object-oriented paradigm because communication is achieved by placing data in a tuple space as opposed to sending messages to objects.

### 2.4 Problem Statement

Neither existing distributed programming languages nor existing middleware solutions deal with *all* hardware phenomena listed above. Some middleware proposals offer partial solutions, but do not fit the object-oriented paradigm. However, the object-oriented paradigm has proven its merit w.r.t. dealing with distribution (and its induced concurrency) because it successfully aligns encapsulated objects with concurrently running distributed software entities [3]. We claim that these observations motivate the need for an Ambient-Oriented Programming paradigm which consists of concurrent distributed object-oriented programming languages offering well-integrated facilities to deal with all the hardware phenomena engendered by mobile networks.

## 3 Ambient-Oriented Programming

In the same way that referential transparency can be regarded as a defining property for pure functional programming, this section presents a collection of language design characteristics that discriminate the AmOP paradigm from classic concurrent distributed object-oriented programming. These characteristics have been described earlier [8] and are repeated in the following four sections.

### 3.1 Classless Object Models

As a consequence of argument passing in the context of remote messages, objects are copied back and forth between remote hosts. Since an object in a class-based programming language cannot exist without its class, this copying of objects implies that classes have to be copied as well. However, a class is – by definition – an entity that is conceptually shared by all its instances. From a conceptual point of view there is only one single version of the class on the network, containing the shared class variables and method implementations. Because objects residing on different machines can autonomously update a class variable of "their" copy of the class or because a device might upgrade to a new version of a class thereby "updating" its methods, a classic distributed state consistency problem among replicated classes arises. Independent updates on the replicated class – performed by autonomous devices – can cause two instances of the "same" class to unexpectedly exhibit different behaviour. Allowing programmers to manually deal with this phenomenon requires a *full* reification of classes and the instance-of relation. However, this is easier said than done. Even in the absence of wireless distribution, languages like Smalltalk and CLOS already illustrate that a serious reification of classes and their relation to objects results in extremely complex meta machinery.

A much simpler solution consists of favouring entirely self-sufficient objects over classes and the sharing relation they impose on objects. This is the paradigm defined by prototype-based languages like Self [34]. In these languages objects are *conceptually* entirely idiosyncratic such that the above problems do not arise. Sharing relations between different prototypes can still be established (such as e.g. traits [33]) but the point is that these have to be explicitly encoded by the programmer at all times[1]. For these reasons, we have decided to select prototype-based object models for AmOP. Notice that this confirms the design of existing distributed programming languages such as Emerald, Obliq, dSelf and E which are all classless.

### 3.2 Non-Blocking Communication Primitives

Autonomous devices communicating over volatile connections necessitate non-blocking communication primitives since blocking communication would harm the autonomy of mobile devices. First, blocking communication is a known source of (distributed) deadlocks [36] which are extremely hard to resolve in mobile networks since not all parties are necessarily available for communication. Second, blocking communication primitives would cause a program or device to block

[1] Surely, a runtime environment can optimise things by sharing properties between different objects. But such a sharing is not part of the language definition and can never be detected by objects.

long-lasting upon encountering volatile connections or temporary unavailability of another device [25, 28]. As such, the availability of resources and the responsiveness of applications would be seriously diminished.

Non-blocking communication is often confused with asynchronous sending, but this neglects the (possibly implicit) corresponding 'receive' operation. Non-blocking reception gives rise to event-driven applications, responsive to the stream of events generated by spontaneously interacting autonomous devices. We thus conclude that an AmOP language needs a concurrency model without blocking communication primitives.

### 3.3 Reified Communication Traces

Non-blocking communication implies that devices are no longer implicitly synchronised while communicating. However, in the context of autonomously collaborating devices, synchronisation is necessary to prevent the communicating parties from ending up in an inconsistent state. Whenever such an inconsistency is detected, the parties must be able to restore their state to whatever previous consistent state they were in, such that they can synchronise anew based on that final consistent state. Examples of the latter could be to overrule one of the two states or deciding together on a new state with which both parties can resume their computation. Therefore, a programming language in the AmOP paradigm has to provide programmers with an *explicit representation* (i.e. a reification) of the communication details that led to the inconsistent state. Having an explicit reified representation of whatever communication that happened, allows a device to properly recover from an inconsistency by reversing (part of) its computation.

Apart from supporting synchronisation in the context of non-blocking communciation, reified communication traces are also needed to be able to implement different message delivery policies. A broad spectrum of such policies exists. For example, in the M2MI library [21], messages are asynchronously broadcasted without guarantees of being received by any listening party. In the actor model on the other hand, all asynchronous messages must eventually be received by their destination actor [1]. This shows that there is no single "right" message delivery policy because the desired delivery guarantee depends on the semantics of the application. Reifying outgoing communication traces allow one to make a tradeoff between different delivery guarantees. Programming languages belonging to the AmOP paradigm should make this possible.

### 3.4 Ambient Acquaintance Management

The combination of autonomous devices and ambient resources which are dynamically detected as devices are roaming implies that devices do not necessarily rely on a third party to interact with each other. This is in contrast to client-server communication models where clients interact through the mediation of a server (e.g. chat servers or white boards). The fact that communicating parties do not need an explicit reference to each other beforehand (whether directly or indirectly through a server) is known as distributed naming [11]. Distributed naming provides a mechanism to communicate without knowing the address of an ambient resource. For example, in tuple space based middleware this property is achieved, because a process can publish data in a tuple space, which can then be consulted by the other processes based on a pattern matching basis. Another example is M2MI [21], where messages can be broadcast to all objects implementing a certain interface.

We are not arguing that all AmOP applications must necessarily be based on distributed naming. It is perfectly possible to set up a server for the purposes of a particular application. However, an AmOP language should allow an object to spontaneously get acquainted with a previously unknown object based on an intensional description of that object rather than via a fixed URL. Incorporating such an acquaintance discovery mechanism, along with a mechanism to detect and deal with the loss of acquaintances, should therefore be part of an AmOP language. We will refer to the combination of these mechanisms as ambient acauqintance management.

### 3.5 Discussion

Having analysed the implications of the hardware phenomena on the design of programming languages, we have distilled the above four characteristics. We will henceforth refer to programming languages that adhere to them as *Ambient-oriented Programming Languages*. Surely, it is impossible to prove that these are strictly necessary characteristics for writing the applications we target. After all, AmOP does not transcend Turing equivalence. However, we do claim that an AmOP language will greatly enhance the construction of such applications because its distribution characteristics are designed with respect to the hardware phenomena presented in section 2.1. AmOP languages incorporate transient disconnections and evolving acquaintance relationships in the heart of their computational model.

## 4 The AmbientTalk Kernel

Having defined the AmOP paradigm, we now present AmbientTalk, a language that was explicitly designed to satisfy its characteristics. As explained in the introduction, AmbientTalk is a reflectively extensible kernel that can be used as a language laboratory to experiment with AmOP language features. First, the essential characteristics of its object model are explained.

### 4.1 A Double-layered Object Model

AmbientTalk has a concurrent object model that is based on the model of ABCL/1 [39]. This model features active objects which consist of a perpetually running thread, updateable state, methods and a message queue. These concurrently running active objects communicate by asynchronous message passing. Upon reception, messages are scheduled in the active object's message queue and are processed one by one by the active object's thread. By excluding simultaneous message processing, race conditions on the updateable state are avoided. The merit of the model is that it unifies imperative object-oriented programming and concurrent programming without suffering from omnipresent race conditions. We will henceforth use the term 'active object' or 'actor' interchangeably for ABCL/1-like active objects.

To avoid having every single object to be equipped with heavyweight concurrency machinery and having every single message to be thought of as a concurrent one, an object model that distinguishes between active and passive (i.e. ordinary) objects is adopted. This allows programmers to deal with concurrency only when strictly necessary (i.e. when considering semantically concurrent and/or distributed tasks). Since passive objects are not equipped with an execution thread, the "current thread" runs from the sender into the receiver, thereby implementing synchronous message passing. However, it is important to ensure that a passive object is never shared by two different active ones because this easily leads to race conditions. AmbientTalk's object model avoids this by obeying the following rules:

– *Containment* Every passive object is contained within exactly one active object. Therefore, a passive object is never shared by two active ones. The only thread that can enter the passive object is the thread of its active container.
– *Argument Passing Rules* When an asynchronous message is sent to an active object, objects may be sent along as arguments. In order not to violate the containment principle, a passive object that is about to leave its active container this way, is passed by copy. This means that the passive object is deep-copied up to the level of references to active objects. Active objects process messages one by one and can therefore be safely shared by two different active objects. Hence, they are passed by reference.

This pragmatic marriage between the functional actor model, the imperative thread model and the prototype-based object model was chosen as the basis for AmbientTalk's distribution model. Active objects are defined to be AmbientTalk's unit of distribution and are the only ones allowed to be referred to across device boundaries. Therefore, AmbientTalk applications are conceived as suites of active objects deployed on autonomous devices. Several active objects can run on a device and every active object contains a graph of passive objects. Objects in this graph can refer to active objects that may reside on any device. In other words, AmbientTalk's remote object references are always references to active objects. The rationale of this design is that synchronous messages (as sent to passive objects) cannot be reconciled with the non-blocking communication characteristic presented in section 3.2.

AmbientTalk does not know the concept of proxies on the programming language level. An active object $\mathtt{a}_1$ can 'simply' refer to another active object $\mathtt{a}_2$ that resides on a different machine. If both machines move out of one another's communication range and the connection is (temporarily) lost, $\mathtt{a}_1$ conceptually remains referring to $\mathtt{a}_2$ and can keep on sending messages as if nothing went wrong. Such messages are accumulated in $\mathtt{a}_1$ and will be transparently delivered after the connection has been re-established. Hence, AmbientTalk's default delivery policy strives for eventual delivery of messages. The mechanism that takes care of this transparency is explained in section 4.4. First we discuss both layers of AmbientTalk's object model in technical detail.

## 4.2 The Passive Object Layer

Following the prototype-based tradition, AmbientTalk passive objects are conceived as collections of slots mapping names to objects and/or methods. The code below shows an implementation for stacks in AmbientTalk:

```
makeStack(size)::object({
  els:makeVector(size);
  top:0;
  isEmpty()::{ size=0 };
  isFull()::{ size=top };
  push(item)::{
    if(this.isFull(),
      { error("Stack Overflow") },
      { top:=top+1;
        els.set(top,item) })
  };
  pop()::{
    if(this.isEmpty(),
      { error("Stack Underflow") },
      { val: els.get(top);
        top:=top-1;
        val })
  }})
```

Objects are created using the `object(...)` form[2]. It creates an object by executing its argument expression, typically a block of code (delimited by curly braces) containing a number of slot declarations. Slots can be mutable (declared with `:`) or immutable (declared with `::`). Mutable slots are always private and immutable slots are always public (for the rationale of this design decision we refer to [9]). Both method invocation and public slot selection use the classic dot notation. Objects are lexically scoped such that names from the surrounding scope can be used in the `object(...)` form. As illustrated by `makeStack(size)`, the form can be used in the body of a function in order to generate objects. Such a function will be referred to as a constructor function and is AmbientTalk's idiom to replace the object instantiation role of classes. Objects can also be created by extending existing ones: the `extend(p,...)` form creates an object whose parent is `p` and whose additional slots are listed in a block of code, analogous

[2] A 'form' is a Scheme-like special form, i.e., a built-in 'function' whose parameters are treated in an ad hoc way. `if(test,exp1,exp2)` is another example of a form.

to the `object(...)` form. Messages not understood by the newly created object are automatically delegated to the parent. Furthermore, a Java-like `super` keyword can be used to manually delegate messages to the parent object. Following the standard delegation semantics proposed by Lieberman [22] and Ungar [34], references to `this` in the parent object denote the newly created child object.

Apart from objects, AmbientTalk features built-in numbers, strings, a null value `void` and functions. However, these 'functions' are actually nothing but methods in AmbientTalk. For example, the `makeStack` constructor function is actually a method of the root object which is the global environment of the AmbientTalk interpreter. Methods can be selected from an object (e.g. `myPush:aStack.push`). Upon selection, a first-class closure object is created which encapsulates the receiver (`aStack`) and which can be called using canonical syntax, e.g., `myPush(element)`. Closure objects are actually passive objects with a single `apply` method. Finally, a syntactic sugar coating allows anonymous closures to be created given a list of formal parameters and a body, e.g., `lambda(x,y) -> {x+y}`. When bound to a name (e.g., as the value of a slot `f` or when bound to a formal parameter `f`), a closure is called using canonical syntax, e.g., `f(1,2)`.

## 4.3 The Active Object Layer

As explained in section 4.1, AmbientTalk actors have their own message queues and computational thread which processes incoming messages one by one by executing their corresponding method. Therefore, an actor is entirely single-threaded such that state changes using the classic assignment operator `:=` cannot cause race conditions. Messages sent to the passive objects it contains (using the dot notation) are handled synchronously. Actors are created using the `actor(o)` form where `o` must be a passive object that specifies the behaviour of the actor. In order to respect the containment principle, a copy of `o` is made before it is used by the `actor` form because `o` would otherwise be shared by the creating and the created actor. A newly created actor is immediately sent the `init()` message and `thisActor` denotes the current actor. These concepts are exemplified by the following code excerpt which shows the implementation of a friend finder actor running on a cellular phone. When two friend finders discover one another (which is explained later on) they send each other the `match` message passing along an `info` passive object that contains objects representing the age (with an `isInRangeOf` method) and hobbies (containing a method that checks whether two hobby lists have anything in common).

```
makeFriendFinder(age,hobbies)::actor(object({
  init()::{ display("Friend Finder initialized!") };
  beep()::{ display("Match Found - BEEP!") };
  match(info)::{
    if(and(age.isInRangeOf(info.age),
           hobbies.intersectsWith(info.hobbies)),
      { thisActor#beep() })
  }}))
```

Actors can be sent asynchronous messages using the # primitive which plays the same role for actors as the dot notation for passive objects. E.g., if `ff` is a friend finder (possibly residing on another cellular phone), then `ff#match(myInfo)` asynchronously sends the `match` message to `ff`. The return value of an asynchronous message is `void` and the sender never waits for an answer. Using the # operator without actual arguments (e.g., `ff#match`) yields a first-class message object that encapsulates the sending actor (`thisActor`), the destination actor (`ff`) and the name of the message. First-class messages are further explained in section 4.5 that describes AmbientTalk's meta-level facilities. Finally, using the dot notation for actors (resp. # for passive objects) is considered to be an error.

When passing along arguments with (both synchronous and asynchronous) message sends, caution is required in order not to breach the containment principle. In the case of synchronous messages of the form $\texttt{po.m(arg}_1\texttt{,...arg}_n\texttt{)}$ between two objects that are contained in the same actor, the arguments do not "leave" the actor and can therefore be safely passed by reference. In the case of asynchronous messages of the form $\texttt{ao\#m(arg}_1\texttt{,...arg}_n\texttt{)}$, the arguments "leave" the actor from which the message is sent. In order to respect the containment principle, this requires the arguments to be passed by copy as explained in section 4.1. In the friend finder example, the `info` object is thus passed by copy.

## 4.4 First-class Mailboxes

AmbientTalk's concurrent object model presented above is classless and supports non-blocking communication. This already covers two of the four characteristics of AmOP as presented in section 3. However, with respect to the other two, the model presented so far still has some limitations which it directly inherits from the original actor model [15, 2]:

– The model does not support the ambient acquaintance management characteristic of the AmOP paradigm because traditionally, actors can only gain acquaintances through other actors. Extensions of the actor model that address this problem (e.g., ActorSpaces [4]) use a centralized authority which is not reconcilable with the hardware phenomena listed in section 2.1.
– Actor formalisms do not support the reified communication traces we argued for in section 3.

To enable these two properties, AmbientTalk replaces the single message queue of the original actor model by a system of eight first-class mailboxes which is described below.

When scrutinising the communication of a typical actor, four types of messages are distinguished: messages that have been sent by the actor (but not yet received by the other party), outgoing messages that have been acknowledged to be received, incoming messages that have been received (but not yet processed) and messages that have been processed. The AmbientTalk interpreter stores each type in a dedicated mailbox associated with the actor. An actor has access to its mailboxes through the names `outbox`, `sentbox`, `inbox` and `rcvbox`. The combined behaviour of the `inbox` and `outbox` mailboxes was already implicitly present in the original actor model in the form of a single message queue. As we will show in the remainder of the paper, AmbientTalk's mailboxes are the fundamental semantic buidling blocks for implementing advanced language constructs on top of the non-blocking communication primitives. Indeed, conceptually, the mailboxes `rcvbox` and `sentbox` allow one to peek in the communication history of an actor. Likewise, the mailboxes `inbox` and `outbox` represent an actor's continuation, because they contain the messages the actor will process and deliver in the future. Together, the four explicit mailboxes cover the need for reified communication traces that have been prescribed by the AmOP paradigm.

In order to cover the ambient acquaintance management requirement of AmOP, AmbientTalk actors have four additional predefined mailboxes called `joinedbox`, `disjoinedbox`, `requiredbox` and `providedbox`. An actor that wants to make itself available for collaboration on the network can broadcast this fact by adding one or more descriptive tags (e.g. strings) in its `providedbox` mailbox (using the `add` operation described below). Conversely, an actor that needs other actors for collaboration can listen for actors broadcasting particular descriptive tags by adding these tags to its `requiredbox` mailbox. If two or more actors join by entering one another's communication range while having an identical descriptive tag in their mailboxes, the mailbox `joinedbox` of the actor that *required* the collaboration is updated with a *resolution object* containing the corresponding descriptive tag and a (remote) reference to the actor that *provided* that tag. Conversely, when two previously joined actors move out of communication range, the resolution is moved from the `joinedbox` mailbox to the `disjoinedbox` mailbox. This mechanism allows an actor not only to detect new acquaintances in its ambient, but also to detect when they have disappeared from the ambient. It is AmbientTalk's technical realisation of the ambient acquaintance management characteristic discussed in section 3.4.

Mailboxes are first-class passive objects contained in the actor. Due to the containment principle for passive objects, mailboxes can never be shared among multiple actors. However, mailboxes are necessarily shared between the actor and the AmbientTalk interpreter because this interpreter puts messages into the mailboxes (e.g. upon reception of a message or upon joining with another actor). To avoid race conditions on mailboxes, the interpreter is not given access while the actor manipulates its own built-in mailboxes[3]. Mailboxes provide operators to add and delete elements (such as messages, descriptive tags and resolutions): if `b` is a mailbox, then `b.add(elt)` adds an element to `b`. Similarly, `b.delete(elt)` deletes an element from a mailbox. `b.iterate(f)` applies the closure `f` to all elements that reside in the mailbox `b`. Moreover, the changes in a mailbox can be monitored by registering observers with a mailbox: `b.uponAdditionDo(f)`(resp. `b.uponDeletionDo(f)`) installs a closure `f` as a listener that will be triggered whenever an element is added to (resp. deleted from) the mailbox `b`. The element is passed as an argument to `f`.

[3] Apart from the eight built-in mailboxes, actors can create their own custom mailboxes which might be used by reflective extensions to temporarily store messages.

The following code excerpt exemplifies these concepts by extending the friend finder example of the previous section with ambient acquaintance management in order for two friend finders to discover each other. The initialisation code shows that the actor advertises itself as a friend finder and that it requires communication with another friend finder. When two friend finders meet, a resolution is added to their `joinedbox`, which will trigger the method `onFriendFinderFound` that was installed as an observer on that mailbox. This resolution contains a `tag` slot (in this case `"<FriendFinder>"`) and a `provider` slot corresponding to the providing actor. The latter is sent the `match` message (as described in the previous section).

```
makeFriendFinder(age,hobbies)::actor(object({
  ...as above...

  onFriendFinderFound(aResolution)::{
    aResolution.provider#match(makeInfo(age, hobbies))
  };
  init()::{
    provided.add("<FriendFinder>");
    required.add("<FriendFinder>");
    joinedbox.uponAdditionDo(this.onFriendFinderFound)
  }}))
```

## 4.5 AmbientTalk as a Reflective Kernel

This section explains how to reflectively extend AmbientTalk's kernel which consists of the double-layered object model along with the system of eight built-in mailboxes. The built-in mailboxes and their observers (installed with `uponAdditionDo` and `uponDeletionDo` as described above) can already be regarded as part of AmbientTalk's metaobject protocol (MOP) since they partially reify the state of the interpreter. Indeed, they constantly reflect the past and future of the communication state between actors as well as the evolving state of the ambient. Additionally, the MOP allows a programmer to override the default message sending and reception mechanisms. Just like ABCL/R [37, 26], AmbientTalk has a MOP for a concurrent active object model (hence what follows is only applicable to active objects, there is no MOP for passive objects). The operations of the MOP presented in this section by default reside in any actor and can be redefined by overriding them in any idiosyncratic actor. This mechanism is at the heart of AmbientTalk's ability to act as a programming language laboratory for AmOP. The remainder of this section describes the different MOP operations.

In order to explain the MOP, it is crucial to understand how asynchronous messages are sent between two actors (that might reside on different machines). When an actor $\mathtt{a}_1$ sends a message of the form $\mathtt{a}_2$`#m(...)`, the interpreter of $\mathtt{a}_1$ creates a first-class message object and places it in the `outbox` of $\mathtt{a}_1$. After having successfully transmitted that message between the interpreter of $\mathtt{a}_1$ and the interpreter of $\mathtt{a}_2$, the interpreter of $\mathtt{a}_2$ stores it in the `inbox` of $\mathtt{a}_2$. Upon receiving a notification of reception, the interpreter of $\texttt{a}_1$ moves the message from the `outbox` of $\texttt{a}_1$ to the `sentbox` of $\texttt{a}_1$. $\texttt{a}_2$ processes the messages in its `inbox` one by one and stores the processed messages in the `rcvbox` of $\texttt{a}_2$. Each stage in this interplay (namely message creation, sending, reception and processing) between the two interpreters is reified in the MOP.

**Message creation** is reified in the MOP with the constructor function `createMessage(sender, dest, name, args)` which generates first-class messages. A message is a passive object which has four slots: the sending actor `sender`, the destination actor `dest`, the name of the message `name` and a vector object `args` containing the actual arguments. Remember from section 4.3 that a first-class message is also created upon field selection with an expression of the form `anActor#msgName` which results in a first-class message with sender `thisActor`, destination `anActor`, name `msgName` and an empty argument vector.

**Message sending** is reified in the MOP by adding messages to the `outbox` which is accomplished by the MOP's message sending operation `send`. In other words, an expression of the form $\texttt{anActor\#msg(arg}_1\texttt{, ..., arg}_n\texttt{)}$ is base-level terminology for an equivalent call to the meta-level method `send`, passing along a newly created first-class message object. The default behaviour of `send` is: `send(msg)::{outbox.add(msg) }`. It is possible to override this behaviour by redefining the method `send`. The example below illustrates how `send` can be overridden for logging purposes.

```
send(msg)::{
  display("sending..."+msg.getName());
  super.send(msg)
}
```

Every actor has a perpetually running thread that receives incoming messages in the `inbox` and transfers them to the `rcvbox` after processing them. **Message reception** is reified in the MOP by adding messages to an actor's `inbox` which can be intercepted by adding an observer to that mailbox. **Message processing** is reified in the MOP by invoking the parameterless `process` method on that message (which will execute the recipient's method corresponding to the message name) and by subsequently placing that message in the `rcvbox`. The latter event can be trapped by installing an observer on that mailbox.

### 4.6 Summary: AmbientTalk and AmOP

In summary, AmbientTalk features a classless double-layered object model. Actors are visible in mobile networks and communicate with each other in a non-blocking way. Internally, they contain a graph of passive objects. Actors have four mailboxes which reify their communication traces and four mailboxes which are causally connected to the outside world to reflect the evolution of acquaintances in the ambient. These properties turn AmbientTalk into an AmOP language as discussed in section 3. AmbientTalk's fundamental semantic building blocks can be used along with the MOP's reflective facilities to experiment with new

AmOP language constructs and their interactions. This is extensively shown in the following section.

## 5 AmbientTalk at Work: AmbientChat

In order to validate AmbientTalk's concepts, we have implemented an instant messenger application that epitomises all the difficulties of mobile network applications in which multiple parties dynamically join and disjoin and collaborate without presuming a centralised server. The instant messenger runs on a mobile device and spontaneously discovers chat buddies appearing in its proximity. Conceived as an actor, the messenger's functionality suggests the following conceptual constructions which are non-existent in the AmbientTalk kernel. Their reflective implementation is the topic of this section:

**Ambient References** can be thought of as active object references which "sniff the ambient" given a textual description (e.g. a nickname). Ambient references discover actors fitting that description and are resilient to the effects of volatile connections: upon disconnection ambient references try to rebind to a (potentially different) actor in the ambient fitting the description.

**Futures** [14, 23] are a classic technique to reconcile return values of methods with asynchronous message sends without resorting to manual callback methods or continuation actors. A future is a placeholder that is immediatly returned from an asynchronous send and that is eventually *resolved* with the expected result. Computations that access an unresolved future block until it is resolved. However, this contradicts the *non-blocking communication* characteristic of AmOP. AmbientTalk's futures avoid this problem by adopting the technique that was recently proposed in E [27]. It allows for a transparent forwarding of messages sent to a future to its resolution and features a `when(aFuture, closure)` construct to register a closure that is to be applied upon resolving the future.

**Due-blocks** are similar to try-catch blocks. They consist of a block of code, a handler and a deadline that is imposed on the transmission of all asynchronous messages sent during the execution of the block. The handler is invoked should the deadline expire. This mechanism is used by the messenger to visualize undelivered messages by greying them out in the GUI.

These language constructs are implemented by using and overriding the MOP operations described in the previous section. We use a mixin-based technique to implement them in a modular way: an AmbientTalk construct and its supporting MOP operations are grouped in what we call a *language mixin*; a function that returns an extension of its argument with new meta-level behaviour (i.e. it overrides `send`, `createmessage`, `process` and/or installs observers on mailboxes). The idea is to apply such a language mixin to a passive object before that passive object is used to create an actor. This way, a newly created actor will exhibit the required behaviour.

Given these language abstractions, the code for the instant messenger follows. An instant messenger in AmbientTalk is conceived as an actor created by the constructor function `makeInstantMessenger` given a `nickname`, a `guiActor` (representing the application's graphical user interface) and a `maxTimeOut` value that indicates how resilient the messenger will be w.r.t. volatile connections. The actor's MOP is extended with the three language constructs by applying the `DueMixin`, the `FuturesMixin` and the `AmbientRefMixin` to the passive object representing its behaviour. The usage of the language constructs is indicated in comments.

```
makeInstantMessenger(nickname, guiActor, maxTimeOut)::
  actor(AmbientRefMixin(FuturesMixin(DueMixin(object({
    buddies : makeHashmap();
    statusLine: "Available";

    getStatusLine() :: { statusLine };
    setStatusLine(newStatus) :: { statusLine := newStatus };
    buddyAppears(buddyNick) :: {
      when(buddies.get(buddyNick)#getStatusLine(),       // FUTURES
           lambda(status) -> { guiActor#onlineColor(buddyNick,status) })
    };
    buddyDisappears(buddyNick) :: {
      guiActor#offlineColor(buddyNick)
    };
    addBuddy(buddyNick) :: {
      bAmsg:thisActor#buddyAppears;   bDmsg:thisActor#buddyDisappears;
      bAmsg.getArgs().set(1,buddyNick);    bDmsg.getArgs().set(1,buddyNick);
      buddies.put(buddyNick,
                makeAmbientRef("<Messenger id="+buddyNick+">", bAmsg, bDmsg))
    };                                                    // AMBIENT REFERENCES
    receiveText(from, text) :: {
      guiActor#showText(from,text)
    };
    failedDelivery(msg) :: {
      text: msg.getArgs().get(2);
      guiActor#unableToSend(text)
    };
    talkTo(buddyNick,text) :: {
      due(maxTimeOut, lambda() -> {                       // DUE BLOCKS
          buddies.get(buddyNick)#receiveText(identity,text)
      }, thisActor#failedDelivery)
    };
    init() :: {
      guiActor#register(thisActor);
      broadcast("<Messenger id="+nickname+">")
    }})))));
```

An instant messenger actor has a slot `statusLine` and a slot `buddies` mapping nicknames to ambient references that represent instant messengers on other mobile devices. Upon creation, its `init` method registers the messenger with the GUI and broadcasts its presence in the ambient. The latter is accomplished by the `broadcast` function which is merely a thin veneer of abstraction to hide the fact that a descriptive tag is added to the `providedbox` of the actor (i.e. `broadcast(tag)::{providedbox.add(tag)}`). Instant messenger actors have three methods (`addBuddy`, `setStatusLine` and `talkTo`) that are called from the GUI when the user adds a buddy (given a nickname), changes his status line or sends a `text` message to one of his buddies. Two other methods (`getStatusLine` and `receiveText`) are invoked by other instant messengers to retrieve a buddy's status line and to send a message to a buddy.

Upon adding a buddy, `addBuddy` creates an ambient reference (which searches for a messenger) based on the nickname and a couple of first-class callback messages (`bAmsg` and `bDmsg`) which are to be invoked by the ambient reference whenever that buddy appears or disappears in the ambient. The first-class callback message `bAmsg` (resp. `bDmsg`) is created by the expression `thisActor#buddyAppears` (resp. `thisActor#buddyDisappears`) and given the buddy's nickname as its first and only argument. Whenever an actor fitting the ambient reference's tag appears (resp. disappears) `buddyAppears` (resp. `buddyDisappears`) will thus be invoked. `buddyAppears` subsequently asks for the status line of its communication partner. This yields a future, that will trigger the `when` language construct upon resolution. In the closure that is passed to the `when` construct, the resolved future can be accessed as a parameter (e.g. `status`). Finally, whenever the GUI invokes `talkTo` to communicate with a buddy, `receiveText` is sent to the ambient reference representing that buddy. It is the ambient reference's task to forward that message to the actual messenger it denotes. The send of `receiveText` occurs in a `due` block which tries to send it within the prescribed time period. Should the message expire, `failedDelivery` is invoked which in turn informs the GUI about this event.

This AmbientTalk application illustrates that it is relatively straightforward to build an AmOP application, given the futures, ambient references and due-block language extensions. The remainder of this section presents their implementation one by one.

## 5.1 Ambient References

As explained above, ambient references are active object references – "pointing into the ambient" – that constantly represent a remote actor fitting some textual description. The following language mixin contains the `makeAmbientRef` constructor function to create an ambient reference actor given a textual description `tag` and two first-class messages `uponJoinMsg` and `uponDisjoinMsg` that need to be sent upon establishing or losing a connection with an actor fitting the description.

```
AmbientRefMixin(actorBehaviour)::extend(actorBehaviour, {
  makeAmbientRef(tag, uponJoinMsg, uponDisjoinMsg)::actor(object({
    principal : void;
    forwardMsg(msg) :: {
      if(not(is_void(principal)), {
        outbox.add(msg.setDestination(principal));
        inbox.delete(msg)
      })
    };
    handleActorJoined(resolution) :: {
      if(is_void(principal), {
        principal := resolution.provider;
        send(uponJoinMsg);
        inbox.iterate(this.forwardMsg)
      })
    };
    handleActorDisjoined(resolution) :: {
      if(resolution.provider.equals(principal), {
        principal := void;
        send(uponDisjoinMsg);
        outbox.iterate(lambda(msg) -> {
          outbox.delete(msg);
          inbox.add(msg)
        })
      });
      disjoined.delete(resolution)
    };
    init() :: {
      requiredbox.add(tag);
      inbox.uponAdditionDo(this.forwardMsg);
      joinedbox.uponAdditionDo(this.handleActorJoined);
      disjoinedbox.uponAdditionDo(this.handleActorDisjoined)
    }}))})
```

The ambient reference is initialised by adding the `tag` to the `requiredbox` making it listen for actors broadcasting this tag, and by installing three mailbox observers to be triggered when messages arrive in the `inbox` and when resolutions appear in the `joinedbox` or `disjoinedbox`. An ambient reference has a private slot `principal`, the value of which is toggled between an actor broadcasting the tag, and `void` when no such actors are currently available in the ambient. This toggling is accomplished by the `joinedbox` observer `handleActorJoined` (called whenever an actor was discovered) and the `disjoinedbox` observer `handleActorDisjoined` (that voids the principal when it has moved out of communication range). When a message is sent to the ambient reference, the `inbox` observer `forwardMsg` is called since it is the ambient reference's task to forward messages to the actor it represents. This is implemented by changing the destination actor of the message from the ambient reference to the principal and by moving it from the `inbox` of the ambient reference to its `outbox`. The AmbientTalk interpreter will henceforth handle the delivery of the message as explained in section 4.5. Messages may be accumulated in the `inbox` of the ambient reference while its `principal` is `void`[4]. Therefore, `handleActorJoined` flushes all unsent messages in the `inbox` by forwarding them to the newly discovered actor. Similarly the `handleActorDisjoined` method will ensure that messages that were not delivered yet and were accumulated in the `outbox` are transferred to the `inbox` of the reference in order to make sure they will be resent upon rebinding to another principal.

## 5.2 Non-blocking Futures

As explained above, AmbientTalk's implementation of futures is based on E. The main difference with existing proposals for futures is the fact that futures are non-blocking. Futures are represented as AmbientTalk actors and messages sent to them are transparently forwarded to the future's value. The `when` construct is used to register a block of code that will be triggered upon resolution of the future. The first part of the language mixin implementing futures is shown below:

```
FuturesMixin(actorBehaviour)::extend(actorBehaviour, {
  makeFuture() :: actor(object({
    value: void;
    whenObservers: makeList();

    forward(msg) :: {
      if(not(has_slot(this, msg.getName())),
        if(is_actor(value),
          { inbox.delete(msg);
            outbox.add(msg.setDestination(value)) }))
    };
    register(aWhenObserver) :: {
      if(is_void(value),
        { whenObservers.add(aWhenObserver) },
        { aWhenObserver.notify(value) })
    };
    resolve(computedValue) :: {
      value:=computedValue;
      whenObservers.iterate(lambda(observer) -> { observer.notify(value) });
      inbox.iterate(this.forward)
    };
    init() :: { inbox.uponAdditionDo(this.forward) }
  }));               // CONTINUED
```

The language mixin introduces the `makeFuture` constructor function which generates new futures. Futures contain a method `forward` to forward messages to the actor it resolved to, except for messages for which the future actor itself has a method slot (such as `register` and `resolve`). Every usage of `when(aFuture, closure)` gives rise to the registration of a 'when-observer object' with the

[4] If an actor has no method to process an incoming message, the default behaviour is to leave it waiting in the `inbox`.

future using `register`. Upon resolution, the future notifies all its registered when-observers and forwards all previously accumulated messages.

To introduce futures in the MOP of actors, `createMessage` is overridden such that asynchronous messages are intercepted to be extended with a `future` slot. Furthermore, the message's `process` method (which will be invoked by the destination actor) is refined in order to resolve the message's future with the computed value. The overridden `send` first performs a super-send to delegate message sending to the default implementation. However, instead of returning `void`, the new implementation returns the future contained in the extended message.

```
                   // FuturesMixin, CONTINUED
createMessage(sender,dest,name,args)::{
  extend(super.createMessage(sender,dest,name,args), {
    future :: makeFuture();
    process()::{
      computedValue: super.process();
      future#resolve(computedValue);
      computedValue
    };
  })
};
send(message)::{
  super.send(message);
  message.future
};

whenBlocks: makeHashmap();
whenIdCtr : 1;

invokeWhen(whenBlockID, computedValue)::{
  whenBlocks.get(whenBlockID)(computedValue);   //curried call
  whenBlocks.delete(whenBlockID)
};
makeWhenObserver(callBackActor, whenBlockID): object({
  notify(computedValue):: {
    callBackActor#invokeWhen(whenBlockID, computedValue) }
});
when(aFuture, whenBlock)::{
  whenBlocks.put(whenIdCtr, whenBlock);
  aFuture#register(makeWhenObserver(thisActor, whenIdCtr));
  whenIdCtr := whenIdCtr + 1
}})
```

The `when(aFuture, closure)` language extension registers a closure that is applied when the future gets resolved. Caution is required since a closure is a passive object and passing it to the future actor would cause it to be copied as a consequence of the containment principle. As this implies deep-copying the entire closure, side effects in the lexical scope would go by unnoticed. Hence, passing closures to another actor must be avoided. This is achieved by creating an observer object (created with the `makeWhenObserver` constructor function) which encapsulates an actor and an ID that identifies a local closure in the `whenBlocks` vector of that actor. It is this observer that is registered with the future. Whenever the future gets resolved, all observers are sent `notify` which in turn ask their encapsulating actor (through `invokeWhen`) to invoke the closure registered on the future by passing along the closure's ID and the result value.

### 5.3 Due: Handling Failures

As explained in section 4.1, AmbientTalk's default delivery policy guarantees eventual delivery of messages. Messages are stored indefinitely in the outbox of an actor until they can be delivered. The `due` language construct alters this policy by putting an expiration deadline on outgoing messages. A `due`-block consists of a timeout value (relative to the time at which a message is sent), a 'body' closure and a handler message to be sent upon expiration. When a message sent during the execution of the body expires, it is removed from the actor's outbox and the handler message is sent with the expired message as argument. The implementation of the `due` language construct consists of two separate language mixins:

- The **DueMixin** defines `due` which stamps all asynchronous messages sent while executing its body with an expiration deadline and a handler message to be sent upon expiration.
- The **ExpiryCheckMixin** makes an actor regularly check its `outbox` in order to remove expired messages and to send their corresponding handler message.

The reason for separating the `DueMixin` and the `ExpiryCheckMixin` is that messages often get forwarded through different actors before reaching their destination. A typical example thereof is when actors are referred to indirectly via an ambient reference as explained in section 5.1: a message may expire in the outbox of the intermediary ambient reference rather than in the outbox of the actor which originally sent the message. Such intermediary actors must therefore be able to detect expired messages even though they do not use the `due` construct. Hence, the `ExpiryCheckMixin` has to be applied to ambient references. This was omitted in section 5.1 for didactic purposes.

The language mixin `DueMixin` is defined as follows:

```
DueMixin(actorBehaviour) :: extend(actorBehaviour, {
  dueTimeout: void;
  dueHandlerMsg: void;
  due(deadline, body, handlerMsg) :: {
    tmpTimeout: dueTimeout;
    tmpHandler: dueHandlerMsg;
    dueTimeout := deadline;
    dueHandlerMsg := handlerMsg;
    value: body();
    dueTimeout := tmpTimeout;
    dueHandlerMsg := tmpHandler;
    value
  };
  createMessage(sender,dest,name,args) :: {
    msg: super.createMessage(sender,dest,name,args);
    if(!is_void(dueTimeout),
      { extend(msg, {
          deadline :: time() + dueTimeout;
          handlerActor :: dueHandlerMsg.getSender();
          handlerName :: dueHandlerMsg.getName()
        }) },
      msg)}})
```

The `DueMixin` installs the `due` construct in an actor and overrides the way its outgoing messages are created in order to stamp those messages by extending them with a `deadline` slot and a ‘complaint address’ in the form of a `handlerActor` and a `handlerName` slot which will determine how to react when the deadline expires. The overridden `createMessage` method first creates a message object `msg` by delegating to the default implementation. Subsequently, `msg` is extended with the slots provided that it was invoked in the dynamic context of a `due`-block (i.e. if `dueTimeout` contains a meaningful value rather than `void`). At any particular time on the execution path of an actor, one active `due`-block exists (cf. try-catch). Its timeout value and its handler are stored in the slots `dueTimeout` and `dueHandlerMsg`. To allow dynamic nesting of `due`-blocks, the current values of `dueTimeout` and `dueHandlerMsg` are saved in temporary variables and are restored upon returning from the `due` body closure.

What remains to be explained is the `ExpiryCheckMixin` that registers a first-class message `notify` with a local clock actor which periodically sends this message. Upon notification, the actor examines outgoing messages stamped with a deadline to check whether they have expired. Expired messages are deleted from the outbox and their handler message is sent to the appropriate actor.

```
ExpiryCheckMixin(actorBehaviour) :: extend(actorBehaviour,{
  pollInterval:1000; // in milliseconds
  init() :: {
    super.init();
    root.clockActor#register(thisActor#notify, pollInterval)
  };
  notify() :: {
    outbox.iterate(lambda(msg) -> {
      if(has_slot(msg, "deadline"), {
        if(time() > msg.deadline, {
           outbox.delete(msg);
           send(createMessage(
                  thisActor, msg.handlerActor,
                  msg.handlerName, makeVector(1).set(1,msg) ))
        })})})}})
```

## 5.4 Evaluation

This section has presented three tentative high-level AmOP language features: ambient references, due-blocks and non-blocking futures. We have adhered to the (functional programming) tradition of modular interpreters to formulate these features as modular semantic building blocks — called language mixins — that enhance AmbientTalk's kernel. AmbientTalk's basic semantic building blocks (consisting of the eight first-class mailboxes, its mailbox observers and its reflective facilities) have been shown to be sufficient to implement these abstractions. The abstractions have been validated in the context of the instant messenger application. Indeed, the essence of communication between two messengers consists of making the corresponding actors get acquainted and in handling the delivery, processing and result propagation of asynchronously sent messages between two autonomous actors that are separated by a volatile connection. To support these different aspects of communication,

**Ambient References** establish and maintain a meaningful connection between two actors on top of a volatile connection. The implementation of ambient references heavily relies on AmbientTalk's ambient acquaintance management facilities (in order to manage the appearance and disappearance of communication partners) as well as its reified communication traces (to flush messages accumulated during disconnection).

**Non-blocking Futures** in combination with the `when` construct allow one to link an asynchronous message send to the code that is to be executed upon result propagation. The `when` construct thus bridges the computational context in which the message was sent and the one in which its result is handled. Furthermore, AmbientTalk's non-blocking futures delay the delivery of received messages until the expected result is ready to receive them. As mentioned in section 3.3, this shows that reified communication traces are at the heart of realigning synchronisation with communication while strictly relying on non-blocking communication primitives as prescribed by the AmOP paradigm.

**Due-blocks** allow the sender to define, detect and deal with permanent disconnections. The `due` language construct shows that although AmbientTalk's default message delivery policy (discussed in section 4.1) implements a resumable communication model (where disconnections are not aligned with failures), one can still cope with permanent failures by reflecting upon an actor's communication traces: by having access to an actor's outgoing message queue which reifies its outgoing messages yet to be delivered, expired messages can be cancelled.

Surely, it is impossible to prove that AmbientTalk's building blocks are necessary and sufficient to cover all future AmOP features. Nevertheless, our analysis in section 3 strongly argues for their necessity and the expressiveness of our reflective extensions detailed in section 5 forms compelling evidence for their sufficiency. Thanks to the abstraction barriers offered by these reusable language constructs, our prototypical messenger application counts merely 35 lines of AmbientTalk code. A chat application with similar goals – called BlueChat [16] – implemented in Java using Bluetooth counts no less than 545 lines of code. BlueChat allows for ambient acquaintance discovery but has no provisions whatsoever to deal with temporarily lost connections.

Currently, a prototype AmbientTalk interpreter was implemented in Java on top of J2ME. It is written in continuation passing style and relies on sockets for inter-device communication over WLAN. Efficiency was not our primary concern in conceiving the implementation. The messenger experiment has been conducted on QTek 9090 cellular phones.

## 6 Conclusion and Future Work

As explained in the introduction, the goal of our research is to a) obtain a better understanding of the structure of future AmOP applications, b) invent expressive programming language abstractions that facilitate their construction and c) get insight in the semantic building blocks lying behind these abstractions in the same vein continuations are the foundations of control flow and environments are at the heart of scoping mechanisms.

Since the conception of AmOP applications is currently still in its infancy, it is hard to come up with good software-engineering criteria for future AmOP language features. That is why our research methology has been based on an unraveling of the hardware characteristics that fundamentally discriminate mobile devices (connected by mobile networks) from classic desktop machines (connected by stationary networks). We have defined the AmOP paradigm as a set of characteristics for programming languages that directly deal with these hardware phenomena in the very heart of their basic computational abstractions.

Instead of merely proposing a number of arbitrarily chosen language features, we have used our analysis of the hardware phenomena to conceive an extensible kernel that comprises a set of fundamental semantic building blocks to implement future AmOP language features. The essence of the semantic experimentarium consists of a double-layered object model, the active objects of which form the basis for concurrency and distribution. Active objects are further equipped with a MOP and a system of eight mailboxes that constantly reflect their computational history as well as the state of the hardware surrounding them. Although it is impossible to prove that these provisions are both necessary and sufficient, we feel that AmbientTalk provides a good basis for further experiments in language design and that the language features proposed here merely scratch the surface of an interesting new branch in distributed computing research. E.g., it remains an open question of how transaction management in classic distributed systems can be transposed to the AmOP setting in which devices holding a lock may leave and never return. Reified communication traces may once again prove useful here, as already exemplified by optimistic process collaboration approaches such as the Time Warp mechanism [17]. Additionally, more insight is required on how to map AmOP features on efficient implementation technology. E.g., new distributed memory management techniques are required because existing techniques are not intended for use in mobile networks.

## References

1. G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. G. Agha and C. Hewitt. Concurrent programming using actors. *Object-oriented concurrent programming*, pages 37–53, 1987.
3. J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.
4. C. J. Callsen and G. Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, 1994.
5. M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, December 2003.
6. L. Cardelli. A Language with Distributed Scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 286–297. ACM Press, 1995.
7. G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):25–33, 2002.
8. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM Press, 2005.
9. T. D'Hondt and W. De Meuter. Of first-class methods and dynamic scope. *RSTI - L'objet no. 9/ 2003. LMO 2003*, pages 137–149, 2003.
10. P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
11. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.
12. IST Advisory Group. Ambient intelligence: from vision to reality, September 2003.
13. M. Haahr, R. Cunningham, and V. Cahill. Supporting corba applications in a mobile environment. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 36–47, New York, NY, USA, 1999. ACM Press.
14. R. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
15. C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.
16. B. Hui. Go wild wirelessly with bluetooth and java. *Java Developer's Journal*, 9(2), February 2004.
17. D. R. Jefferson. Virtual time. *ACM TOPLAS*, 7(3):404–425, 1985.
18. A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, 1997.
19. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

20. K. Kahn and Vijay A. Saraswat. Actors as a special case of concurrent constraint (logic) programming. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 57–66, New York, NY, USA, 1990. ACM Press.
21. A. Kaminsky and H.-P. Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 72–73, New York, NY, USA, 2002. ACM Press.
22. H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 214–223. ACM Press, 1986.
23. B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.
24. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, page 263, Washington, DC, USA, 2004. IEEE Computer Society.
25. C. Mascolo, L. Capra, and W. Emmerich. Mobile Computing Middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.
26. H. Masuhara, S. Matsuoka, and A. Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. In *Proceedings of Reflection Symposium '96*, pages 79–91, April 1996.
27. M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.
28. A. Murphy, G. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.
29. P. Reynolds and R. Brangeon. DOLMEN - service machine development for an open long-term mobile and fixed network environment. 1996.
30. A. Schill, B. Bellmann, W. Bohmak, and S. Kummel. Infrastructure support for cooperative mobile environments. *Proceedings of the Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE '95*, pages 171–178, 1995.
31. K. Taura, S. Matsuoka, and A. Yonezawa. Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pages 275–292, 1994.
32. D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer. The case for non-transparent replication: Examples from Bayou. *IEEE Data Engineering Bulletin*, 21(4):12–20, december 1998.
33. D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing programs without classes. *Lisp Symbolic Computing*, 4(3):223–242, 1991.
34. D. Ungar and R. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.
35. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, 2001.

36. C. Varela and G. Agha. What after java? from objects to actors. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 573–577, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.
37. T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 306–315. ACM Press, 1988.
38. M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.
39. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.
40. S. Zachariadis, L. Capra, C. Mascolo, and W. Emmerich. Xmiddle: information sharing middleware for a mobile environment. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 712–712, New York, NY, USA, 2002. ACM Press.