Ambient-Oriented Programming

Jessie Dedecker* Tom Van Cutsem*
Stijn Mostinckx† Wolfgang De Meuter Theo D'Hondt
Programming Technology Laboratory
Department of Computer Science
Vrije Universiteit Brussel, Belgium
jededeck | tvcutsem | smostinc | wdmeuter | tjdhondt@vub.ac.be

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 such mobile networks is extremely hard in conventional programming languages because the network is dynamically defined. This hardware phenomenon leads us to postulate a suite of characteristics of future *Ambient-Oriented Programming* languages. A simple reflective programming language kernel, called AmbientTalk, that meets these characteristics is subsequently presented. The power of the reflective kernel is illustrated by using it to conceive a collection of high level tentative ambient-oriented programming language features.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications—*distributed languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms
Design, Languages

Keywords
ambient intelligence, mobile networks, actors, language kernel

1. INTRODUCTION
Applications for mobile devices are given a new meaning with the advent of *mobile networks*. Mobile networks surround a mobile device with wireless technology and are demarcated dynamically as users move about. Mobile networks turn the applications running on mobile devices from mere isolated programs into smart applications that can cooperate with their environment. As such, mobile networks take us one step closer to the world of ubiquitous computing envisioned by Weiser [48]; a world where (wireless) technology is gracefully integrated into the everyday lives of its users. Recently, this vision has been termed *Ambient Intelligence* (AmI for short) by the European Council's IST Advisory Group [27].

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 the 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 low-level system software and networking libraries that provide uniform interfaces to these wireless technologies (such as JXTA [23] and M2MI [31]) have matured in the last couple of years, developing application software for mobile networks still remains difficult. One of the main reasons for this is that current-day programming languages lack abstractions to deal with the mobile hardware characteristics. Indeed, in traditional programming languages failing remote communication is usually dealt with using the classical exception handling mechanism. However, solutions based on classical exception handling do not suffice anymore because potential mobile network failures are the rule rather than the exception. This results in the entire application code being polluted with exception handling code. This observation justifies the need for a new *Ambient-Oriented Programming* (AmOP for short) paradigm that consists of programming languages that explicitly incorporate potential network failures in the very heart of their basic computational steps.

One of the problems is that the AmOP language paradigm forms a moving target because the field is relatively new and only few applications have been developed that start to explore the potentials of mobile networks. Therefore, our research has focussed on the hardware phenomena that distinguish mobile networks from existing stationary networks, in order to distill a number of fundamental programming language characteristics that define the AmOP paradigm. These characteristics were used to define a reflective AmOP language kernel called AmbientTalk. AmbientTalk is presented as an improvement on the classical actor model in the sense that it meets the characteristics prescribed by the AmOP paradigm whereas the classical actor model does not.

*Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)
†Author funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen"

AmbientTalk as a kernel is too rudimentary to be used as a practicable programming language, but its building blocks have been designed in a way that it can be used to further explore the language design space opened up by the described hardware evolution. We show this by using the reflection operators of the language in order to build a collection of tentative high level AmOP language features that facilitate programming for mobile networks.

The discourse of this paper is organised as follows. In the next section we begin by listing the fundamental hardware phenomena that distinguish mobile networks from existing technology. We also situate our research in the field of existing programming languages and middleware in order to motivate AmOP. From these phenomena, section 3 derives a collection of programming language characteristics we will use to define the AmOP paradigm. Based hereupon, the reflective AmbientTalk kernel language is presented in section 4. Finally, in section 5 we show how AmbientTalk enables the exploration of the AmOP programming language design space by presenting a set of language constructs aimed at high level application development for mobile networks.

2. MOTIVATION

In the context of mobile networks, one sometimes makes a distinction between nomadic and ad hoc distributed systems depending on whether a shared infrastructure is used to support the mobile communication. Nomadic systems use such an infrastructure (e.g. cellular phones hopping from one cell to another) whereas ad hoc ones do not (e.g. two PDA's that encounter each other). Figure 1 depicts a scenario where multiple mobile devices are communicating using contemporary infrastructure for mobile communication. The infrastructure is typically used to extend the communication range of mobile devices. Both types of distributed systems have different requirements with respect to their behaviour in isolation and to the scalability [36], but many characteristics are shared.

The technical hardware properties of the devices which constitute ad hoc and nomadic distributed systems engender a number of phenomena that have to be dealt with by the software support (i.e. middleware and/or distributed language processors) that one uses for application programming on the devices. We summarize these hardware phenomena in section 2.1 and describe how existing support under the form of both programming languages and middleware fail to deal with them in sections 2.2 and 2.3. This forms the main motivation for our work.

2.1 Hardware Phenomena

With the current state of commercial technology, mobile devices are often characterised by scarcer resources (such as lower CPU speed, smaller memory and limited battery) than traditional hardware. However, we cannot help but notice that in the last couple of years, mobile computing devices and "real computers" like laptops are blending more and more. That is why we do not consider these restrictions as fundamental to the AmOP paradigm as we consider the following considerations to be:

- **Connection Volatility**
  Two processes that perform a meaningful task together on two cooperating devices 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 a broken connections. However, upon restoring 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 the 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. We say that the available resources are now ambient.
- **Autonomicity**
  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, and especially in mobile ad hoc networks, a connection to such a "higher authority" is not always possible. Every device acts 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 might wait for the results of a request to the server in order to resume its computation. Hence, in theory a distributed system is not necessarily a concurrent one. However, even in the extreme case where both parties of communicating mobile devices run a single threaded program, the autonomicity implies that the resulting task is a concurrent one. Moreover, the trend of software getting ever more multithreaded will also manifestate itself on mobile devices. As a result, concurrency is a natural phenomenon in software running on mobile networks.

2.2 Languages

To the best of our knowledge no distributed language has been specifically designed to meet the harsh characteristics of mobile computing just described. The distributed languages that have been proposed in the past can be categorized as languages designed for local area networks and languages that have been designed for open networks, such as the internet.

2.2.1 Languages for Local Area Networks

A number of distributed languages have been proposed, which target local area networks. Some of them are based on synchronous (hence blocking) communication primitives, such as Emerald [29] and Obliq [10], while others, like the ABCL/f [44] and Argus [33] promote an intermediate form based on futures. These communication mechanisms are feasible for reliable networks, where failures are the exception, but harm the autonomicity when used in high latency networks, such as mobile networks.

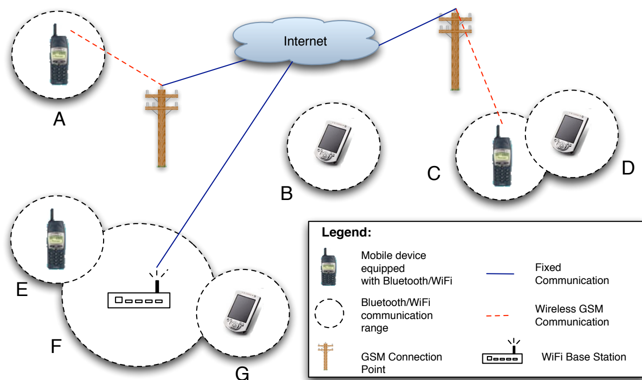

Figure 1: Communication Infrastructure. A communicates with C through GPRS. C and D form an ad hoc network via Bluetooth. B is isolated. E and G communicate with each other via F.

2.2.2 *Languages for Open Networks*

Some distributed languages, such as Janus [30], Salsa [46] and E [39], are based on the actor model. The actor model is based on pure asynchronous communication, which preserves the autonomicity of devices in the context of high latency and failures. However, these languages offer no support to discover ambient resources or to deal with consistency issues found in partitioned networks.

2.3 Middleware

An alternative to distributed languages is middleware. Over the past few years a lot of research has been invested in middleware for nomadic and ad hoc distributed systems [36]. This bulk of research can be categorized into several groups.

2.3.1 *RPC-Based Middleware*

Alice [24] and DOLMEN [41] are attempts to make CORBA feasible for supporting nomadic distributed systems. These attempts focussed mainly on making heavyweight ORBs suitable for the lightweight devices and on improving the resilience of the IIOP protocol to failing communication. Other approaches adapt the RPC protocol by supporting queuing of RPCs [28] or enabling rebinding of resources [43]. These approaches work well when connections are lost for a short time, but do not address disconnections over longer periods of time.

2.3.2 *Publish-Subscribe Middleware*

Another, more recent, branch of middleware for mobile computing is based on the adaptation of the publish-subscribe paradigm [20] to cope with the characteristics of mobile computing [13, 9, 14]. Such middleware allows to express context-aware communication, but has the disadvantage that communication has to be done via callbacks. Such callbacks clutter the code and make the program less understandable.

2.3.3 *Tuple Space Based Middleware*

In the past few years middleware has been proposed [40, 15, 35, 21] for mobile computing based on tuple spaces [22]. A tuple space is used as an intermediary data structure to coordinate communication between processes. This intermediary data structure enables communication that is uncoupled in both time and space. These two properties of communication are interesting in the context of mobile computing. Time uncoupling enables communication when two processes are not available at the same time (for example, when a party is not connected or when it is processing data). The uncoupling in space enables undirected communication between processes, which is also important in the context of mobile computing as a means to discover communication partners in the ambient. Most of the research done so far on tuple spaces in the context of mobile computing can be categorized as efforts to distribute the tuple space over a set of devices.

Although tuple spaces are an interesting communication paradigm for mobile computing, the paradigm does 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. Also, most tuple space based middleware do not provide the means for expressing failure semantics.

2.3.4 *Data Sharing-Oriented Middleware*

Another branch of middleware tries to maximize the autonomicity of mobile devices. In most approaches, such as Coda [42], Bayou, [45], Rover [28] and XMiddle [49], this is achieved by introducing weak replica management facilities in the middleware. Due to the connection volatility replica's are not always synchronized. This can lead to a series of conflicts, which are often application specific and must be resolved depending on the application.

2.4 Summary

The current state of the art in middleware does not address all the important characteristics that are encountered when developing a nomadic or ad hoc distributed system. Regarding distributed languages there have been a number of proposals suitable for open distributed networks, but they have not yet been applied in the context of mobile computing and do not consider characteristics such as service discovery and limited resources.

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 define the boundaries of the ambient-oriented programming paradigm. These characteristics are directly derived from the hardware phenomena we summarized in section 2.1. Until now, it seems that the object-oriented paradigm is the most successful one w.r.t. dealing with distribution and its induced concurrency because it successfully aligns encapsulated objects with concurrently running distributed software entities [6]. Therefore, our most basic assumption is that ambient-oriented programming languages necessarily are concurrent distributed object-oriented programming languages. However, ambient-oriented programming languages differ from other distributed concurrent object-oriented programming languages in the following ways:

3.1 Non-Blocking Communication Primitives

The fact that every hardware device is an autonomous computational entity (inducing natural concurrency) combined with the fact that connections are volatile, implies the necessity for non-blocking communication primitives. Blocking communication is a source of (distributed) deadlocks [47]. Deadlocks and distributed deadlocks in local networks are not considered to be that harmful, since the cause of the deadlock can be easily debugged with contemporary remote debugging environments. However, in mobile networks, not all parties are necessarily available for communication and this makes the resolution of deadlocks extremely hard in such networks. Another, more important consideration when designing a concurrency model for a language that is to run on mobile networks, is that the mechanism should minimize the duration resources are locked. This is very important, because the extremely high latency of communication (over volatile connections) in mobile networks would diminish the availability of resources. Indeed, having blocking communication primitives would imply a program or device to block upon encountering unstable connections or temporary unavailability of another device. This has previously been remarked on several occasions [36, 14, 40]. We thus conclude that an ambient-oriented concurrency model is a concurrency model without blocking communication primitives.

Quite often, the issue of non-blocking communication is confused with asynchronous communication. It is important to see that asynchronous message sending is only one half of non-blocking communication. Asynchronous communication implies that the send operation is non-blocking. However, asynchronous communication tells us nothing about the receive operation (which might be implicitly present). A typical example of asynchronous send operations combined with blocking receive operations is found in the tuple-space based middleware (discussed in section 2.3.3), which provide explicit, blocking receive operations on the tuple-space .

3.2 Reified Communication Traces

Non-blocking communication (both send and receive) combined with the autonomicity of the communicating devices implies that they will have to foresee *some* form of handshaking given the fact that these devices are performing a meaningful task together. Since the communication is non-blocking, both senders and receivers will continue their execution irrespective of what happened after a message send. This means that the parties might end up in a state that is not consistent with the semantics of whatever the task it is that they are solving. 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 decide what to do based on the final consistent state they agreed upon. Examples of the latter could be overruling one of the two computations or deciding together on a new state with which both parties can resume their computation. Therefore, a programming language in the ambient-oriented paradigm will have to provide us with reversibility provisions giving programmers a way to manipulate their execution state based on an *explicit representation* (i.e. a reification) of the communication details that led to the inconsistent state. The explicit representation will allow them to take the appropriate actions to reverse (part of) the computation. Notice that any implicit way to prevent the communicating parties from ending up in an inconsistent state implies that communication primitives are blocking, which was precluded above. 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.

Several degrees of message delivery guarantees can be associated with non-blocking communication. For example, in the many-to-many invocations library [31], where all communication occurs via asynchronous messages, there are no delivery guarantees. When a message is sent and there is no process listening for messages, the message is lost. Such communication paradigm is light-weight with respect to the usage of resources and is suitable when no delivery guarantees are to be met. On the other end of the spectrum there is the actor model, where all asynchronous messages that are sent must eventually be received [1]. Such an approach is perhaps feasible when there are abundant resources, but in the context of mobile computing, where devices have scarce resources, it is clear that such an approach is not practicable. This shows that there is no single “right” message delivery guarantee policy because a tradeoff will have to be made based on the requirements of the application and on the available resources. Programming languages belonging to the ambient oriented paradigm should make this tradeoff possible instead of imposing a single strategy. Explicit control over the communication traces allows one to make the tradeoff between different delivery guarantees.

3.3 Ambient Acquaintance Management

The fact that hardware devices are autonomous, combined with the fact that resources are dynamically detected as the devices are roaming means that all devices potentially have the same capabilities to interact with each other directly without relying on a third party. This is in contrast to client-server communication models where clients usually interact through the mediation of a server (such as is the case with chat servers or white boards). The fact that communicating parties do not need to have an explicit reference to each other (whether directly or indirectly through a server) requires what is known as distributed naming [22]. 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 many-to-many invocations [31], where broadcasts to all objects implementing a certain interface can be expressed. Distributed naming is especially important in the context of ad hoc distributed systems, because it provides a mechanism to communicate when the addresses of the processes are not known beforehand.

We are not arguing that all ambient-oriented applications must be based on distributed naming. It is perfectly possible that a programmer (or even a suite of running processes) sets up a server for the purposes of a certain application. However, an ambient-oriented programming language should allow applications to rely on distributed naming if it is required. In other words, the acquaintances of an object must be dynamically manageable).

3.4 Discussion

The three characteristics introduced above define ambient-oriented programming languages. It is hard to prove that these three characteristics for future languages are necessary and sufficient to support future ambient applications. Since the characteristics were derived from the hardware phenomena presented in section 2.1, they are clearly necessary. The fact that they are probably also sufficient is motivated by the fact that in order to send a message, one must a) establish an acquaintance relation between communicating parties, b) have primitives to exchange the message, and c) have primitives to manipulate the message. These aspects are all covered by the above considerations.

The current state of the art in distributed languages does not conform to the characteristics of AmOP. On the one hand, languages for local area networks do not meet the non-blocking communication characteristic. On the other hand, languages for open networks usually satisfy the non-blocking communication characteristic, but do not allow for ambient acquaintance management and do not provide reversibility provisions.

Cardelli and Gordon have proposed a formal model, named mobile ambients, which was designed to make an abstraction of mobile computations and mobile computing [11]. Our approach is more pragmatic, as it deals with the issues directly at the programming language level.

4. THE AMBIENTTALK KERNEL

Now that we have established the characteristics of the ambient paradigm, the stage is set for the introduction of the kernel language AmbientTalk, which enables language designers to explore the realm of language features that facilitate ambient-oriented programming. Before explaining the essential characteristics of the object system of the language, we introduce the ambient actor model, which lies at the heart of the kernel language. Afterwards we discuss the specific language features and reflective operators of AmbientTalk that enable language extensions.

4.1 The Ambient Actor Model

The ambient actor model [19] is an extension of the actor model, which was first proposed by Hewitt [26] and further developed by Agha [2]. The actor model behaves well with regard to the non-blocking communication characteristic of the paradigm. This is because all communication between actors (the unit of concurrency and distribution) is asynchronous and the model does not introduce an explicit receive operation. However, the model has some limitations with regard to the other two characteristics.

- **Ambient Acquaintance Management**: The model does not support the *Ambient Acquaintance Management* characteristic of the ambient paradigm, because an actor relies on other actors to gain new acquaintances. The ActorSpace model [8], an extension to the actor model, enables distributed naming by introducing an actor grouping mechanism, named *spaces*. However, these spaces are managed by centralized authorities, which is infeasible in a mobile computing setting. Furthermore, it is not specified how the ActorSpace model behaves if an authority manipulates the space if the network is partitioned, which is a common occurance in mobile networks.
- **Reified Communication Traces**: Actors and ActorSpaces do not support reified communication traces, because the model guarantees eventual delivery. As a consequence, it is e.g. impossible to introduce language constructs which retract messages that were sent, but not yet transmitted. This kind of flexibility is sometimes needed to resolve conflicts in the case of network partitions as argued in section 3.2.

The Ambient Actor Model [19] alleviates these restrictions with the introduction of *explicit* mailboxes. The use of such mailboxes is twofold: making the communication state of an actor explicit and allowing for ambient acquaintance management. Both uses are detailed below.

4.1.1 Communication State

The ambient actor model provides explicit control over the communication state of an actor. When scrutinising the communication structure of the actor model, we can distinguish between four types of messages. The first type of messages are those an actor received but still needs to process. A second type of messages are those the actor has sent but that have not yet been transmitted. Third, there are messages that an actor has received and processed. Finally, there are the messages that an actor has sent and transmitted. Together, these four types of messages describe the complete communication trace of an actor over time.

In contrast to the regular actor model, where every actor merely has an implicit message queue for accumulating incoming and outgoing messages, the ambient actor model allows clear distinction of these four types of messages by introducing four explicit mailboxes. The messages of the first type are put in the mailbox `in`, the second type of messages are put in the mailbox `out`. If an actor receives a message, then that message will be put in the mailbox `in`, waiting to be processed by that actor. When a message is sent by an actor it is put in its mailbox `out`, waiting to be transmitted to the recipient of that message. Both mailboxes `in` and `out` are implicitly present in the actor model and enable non-blocking communication primitives, which are a necessary characteristic for the ambient-oriented programming paradigm as argued in section 3.1.

In addition to the mailboxes `in` and `out` there are two more mailboxes, `rcv` and `sent`, for the third and fourth type of messages respectively. In the ambient actor model, when a message is processed it is moved from the mailbox `in` to the mailbox `rcv` and when a message is actually transmitted to another actor, then the message is moved from the mailbox `out` to the mailbox `sent`. Conceptually, the mailboxes `rcv` and `sent` allow one to have a peek in the past of the communication history of an actor. Note that the mailboxes `in` and `out` of the actor represent its continuation, because these two mailboxes contain the messages it will process and transmit in the future. Hence, through the introduction of these four explicit mailboxes we have a gate to the past and the future of the actor's state of communication, which enables the reified communication traces that were argued in

section 3.2.

Apart from the four mailboxes that control the state of communication, every actor can create custom mailboxes. Messages can reside in multiple mailboxes at the same time. The status of the delivery of a message can be monitored and altered by accessing the appropriate mailbox. For example, by removing a message from the mailbox `out` we can stop the message from being delivered. Hence, by giving access to the mailboxes, first-class continuations are attained. The mailboxes `in` and `out` not only allow one to have a peek in the future computation and communication of the actor, but even to manipulate it. For example, we could remove a message from the mailbox `in` and thereby prevent it from being processed by the actor.

4.1.2 Ambient Acquaintance Management

In section 3.3 we argued that a form of distributed naming should be possible in ambient-oriented programming languages. In the AAM, distributed naming is available via a pattern-based lookup mechanism. A pattern is an abstract description of a set of actors and is specified by a communicable value. An actor that wants to search for certain other actors in its ambient places a corresponding pattern in its mailbox `required`. Conversely, when an actor wants to make itself available for other actors it places a pattern with a description of itself in its mailbox `provided`. In the former case the actor is said to require a pattern, while in the latter case the actor is said to provide a pattern. Multiple patterns can be added to a mailbox so that an actor can require or provide multiple patterns simultaneously. A pattern can also be removed from either mailboxes at any time when the actor no longer requires or provides a certain pattern.

When two or more actors enter one another's communication range and have a corresponding pattern in their mailboxes, the mailbox `joined` of the actor that required the pattern is updated with a *resolution*. Such a resolution is a pair consisting of the pattern and a reference to the actor who provided the pattern. Conversely, when two actors with a corresponding pattern in their mailboxes are pulled out of communication range, the resolution is moved from the mailbox `joined` to the mailbox `disjoined`. This mechanism allows actors not only to detect new resources in its ambient, but also to detect when actors have disappeared from the ambient. Through this mechanism an actor can manage the acquaintances it encounters in its ambient, which is a characteristic required for ambient-oriented programming languages as discussed in section 3.3

4.1.3 Mailbox Properties

The operational semantics of the ambient actor model [19] exhibits two mailbox properties which are important to avoid race conditions on the mailboxes of an actor.

1. Mailbox Privacy
   Each mailbox has a unique name within an actor. A mailbox is associated with exactly one actor and an actor cannot communicate a reference to one of its mailboxes [1]. Hence, mailboxes are never shared among multiple actors. This is called the *mailbox privacy* property.

2. Serial Mailbox Access
   In the ambient actor model a mailbox is manipulated by two different entities: the actor owning the mailbox and the actor system which updates the mailbox when certain low-level events occur, for example when a message is transmitted. The operational semantics of the ambient actor model is defined in such a way that the manipulation of mailboxes by these two entities cannot occur concurrently. Hence, while an actor processes a message its mailboxes can only be changed by itself, not by the actor system. Messages that the actor system cannot send at that time remain in the `out` mailboxes of the corresponding actors until they can be transmitted. The characteristic that only one entity can manipulate a mailbox at a time is called the *serial mailbox access* property.

Both the mailbox privacy and the serial mailbox access properties are important, because they preserve the encapsulation of the actors and avoid race conditions on mailboxes.

4.2 The Object System

The object model of AmbientTalk is based on the object model of Pic% [17] (pronounced as Pic-oh-oh), which is a small multi-paradigm (functional and object-oriented) programming language. The language is dynamically typed, based on prototype objects and was designed following the principles of little languages [5]. The code below shows the implementation of the prototypical boolean objects:

```
true: object({
  new()::{ this() };
  ifTrue(code())::{ code() };
  not()::{ false };
  and(exp())::{ exp() };
  or(exp())::{ this() }
});

false: object({
  new()::{ this() };
  ifTrue(code())::{ void };
  not()::{ true };
  and(exp())::{ this() };
  or(exp())::{ exp() }
});

test: random(true, false);
test.ifTrue({
  display("The value is true.", eoln)
})
```

Methods are invoked using the dot-operator. Names are declared using either `:` or `::`. The former declares variables, while the latter declares constants. These two types of declarations are also aligned with the scoping rules of the slots in an object [17], but the details for this would lead us too far for the purpose of this paper.

One of the key features that makes AmbientTalk an extensible language is its special call-by-name parameter passing technique [18]. This mechanism is used later in the

[1] However, it is possible to communicate the name of a mailbox.

paper to introduce some of the language constructs. The call-by-name mechanism allows one to specify which actual parameters should be evaluated lazily. The unevaluated expression, which is provided as an actual parameter, is bound to the call-by-name formal parameter and forms a first-class function. In the example, the implementation of the method `ifTrue` has a formal call-by-name parameter `code()`. When an object calls the method `ifTrue`, with the expression `{display("The value is true.", eoln)}` as its argument, then the expression is evaluated to a first-class function `code()` having that expression as its body. This process of thunkification occurs automatically when the formal parameter of the method has an arguments list[2]. Invoking `code()` in the body of the method `ifTrue` (such as in the version of the `true` object) evaluates the expression that was provided as the actual parameter of the method.

4.3 Integrating the Actor Model

In order to create an actor-based language there is a need for three basic ingredients: a construct to create new actors, another construct for message sending and finally a construct to change the state and behavior of the actor. These three constructs are briefly discussed using the following subsections. They are explained throughout the following example:

```
counter: object({
  n: void;
  new(aNumber)::{ copy(n:=aNumber) };
  increment()::{ become(counter.new(n+1)) };
  decrement()::{ become(counter.new(n-1)) };
  get(customer):: { customer<-result(n) };
  init()::{ display("initialized as actor") }
});

mycounter: actor(counter.new(5));
mycounter<-increment();
mycounter<-decrement()
```

4.3.1 *Creating new actors*

In AmbientTalk a new actor is created with the `actor` primitive. This primitive takes one argument, that of an object and returns a reference to the newly created actor with the object as its behaviour. After the actor has been created a message `init` is sent to that actor, which can be used to initialize it. The object provided to the primitive `actor` is passed by copy to ensure that no data is shared by two actors. For example, the expression `actor(counter.new(5))` wraps a clone of the counter object in an actor entity.

4.3.2 *Message sends*

Messages can be sent to an actor by using the `<-` operator. The expression `mycounter<-increment()` sends an asynchronous message `increment` to the actor referred to by the variable `mycounter`. The return value of the asynchronous message is `void`. The arguments of a message are passed by copy in the case of regular objects and primitive values except for actors which are passed as a reference.

4.3.3 *Changing the State and Behavior*

In AmbientTalk the state and behavior of the actor can be changed with the statement `become`. The `become` takes one argument, an object that will process future messages. The argument is passed by copy to prevent data sharing. In the example above, the state of the counter is updated using the `become` primitive after an `increment` or `decrement` message with a clone of the counter object containing the updated state. An alternative to alter the state of an actor in AmbientTalk is to use assignments in the `increment` and `decrement` methods (e.g. `increment()::{n:=n+1}`).

The `become` primitive[3] can also be used to change the behavior of an actor at run-time by providing an object with other behavior as an argument. When a message is sent to an actor and that actor currently does not have any behavior to process that message then that message is kept in the mailbox `in` of the actor. When the behavior is changed into a new behavior that contains behavior for that message, then it gets processed.

4.4 Reflection

Now that we have introduced the basic actor operations in the language a metaobject protocol [38, 37, 12] on top of the AAM is introduced to reify communication between actors.

The default reflective operators presented in this section reside in the root object, which all objects stem from. The reflective operators can be redefined by overriding them in the same root object or in a more specific object. In the former case the redefined reflective operators will change the behaviour of all actors in the system. In the latter case the redefined reflective operators will only affect the actors defined with the specific object as their behaviour. Details on the object-based inheritance model can be found in [17].

4.4.1 *Messages*

Similar to the ambient actor model messages are represented as first-class entities. A message has four required attributes: the source, which is a reference to the actor that sends the message, the target, which refers to the receiver of the message, the name of the method, which should be invoked when the message is processed and finally the arguments for that method. Optionally, it is also possible to *attach* additional information to the message. The attachments are used to pass information to the meta-level interfaces of other actors. Their use becomes clear in the examples in the next section. The function `createMessage` is used to create new messages and takes a minimum of four arguments:

```
createMessage(src, dest, name, args, att1, ..., attN)
```

The attached information can be retrieved with the function `attachment` that takes a message and a number as an argument.

A message can also be created using field selection with the expression `anActor<-msgName`. The expression results in a first-class message with name `msgName` targeted to `anActor`. If the variable `msg` refers to such a message, then evaluating `msg(arg1, ..., argN)` will send the message with the

[2]This is as opposed to manual thunkification in languages such as Smalltalk or Self, where the expressions that are to be lazily evaluated have to be placed in a block at the site of the caller.

[3]To preserve an actors encapsulation, an actor can only become behaviour that the actor itself has defined [3].

arguments arg1 to argN to anActor.

4.4.2 Mailboxes

In the AmbientTalk language mailboxes behave as described in the ambient actor model, discussed in section 4.1. These mailboxes are represented as clones of the prototype mailbox. A mailbox has methods to add, remove and retrieve its contents. The eight mailboxes that give control over the communication state and actor lookup can be accessed in the context of an actor using their designated identifiers[4].

It is possible to observe changes that are made to the mailboxes by registering listener messages, e.g. aMailbox.addListener(aMessage). If aMailbox is altered then these registered listener messages are placed in the outbox of the actor to which the mailbox belongs with the element that was changed in the mailbox as an argument.

Suppose that the messages in and rcv are respectively subsribed as listeners for additions to the inbox and rcvbox, then the following declarations:

```
in(msg)::{ display("received ", msgName(msg), eoln) };
rcv(msg)::{ display("processed ", msgName(msg), eoln) }
```

allow us to write metaprograms based on events of the communication between actors. The notification of changes in the mailboxes occurs in an asynchronous fashion so that this mechanism can be used to write event-based asynchronous metaprograms in the language as illustrated above.

The use of mailbox observers averts the introduction of a blocking accept[5] and transmission operator at the meta-level. The introduction of these operators would allow a developer to design blocking communication abstractions. Hence, through the use of mailbox observers one can still intervene in the communication process at the meta-level, while precluding the implementation of blocking communication constructs via the meta-level interface.

4.4.3 Communication

The kernel can also be used to specify synchronous behaviour to extend the semantics of message sending and message processing between actor entities.

Message Sending

The default behaviour of message sending between actors is defined by the following method defined in the root object:

```
send(target, msgName, args)::{
  msg: createMessage(thisActor(),target,msgName,args);
  outbox.add(msg);
  void
}
```

An expression of the form anActor<-msg(arg1, ..., argN) is mere syntax for a call to the meta-level method send. For example, the expression mycounter<-increment() is translated to the call send(mycounter, "increment", []).

As illustrated below, it is possible to override the default behaviour by redefining the method send with new behaviour. The example above displays "before send" and "after send", respectively before and after sending the message.

```
send(target, msgName, args)::{
  display("before send", eoln);
  super().send(target, msgName, args);
  display("after send", eoln)
}
```

Message Processing

The default behaviour of an actor for processing a message is defined with the following method:

```
process(message)::{
  execute(message)
}
```

The native execute method invokes the method associated with the name of the message in the behaviour of the actor. The process method can be overridden just like the send method.

Note that all the reflective operators in the language (namely, overriding send and process and the explicit manipulation of mailboxes based on listeners) are aligned with the communication mechanisms of the AAM. The send and process are realized through the use of the mailboxes outbox and inbox respectively. In a similar way mailbox observers are invoked via these communication mechanisms. This is important in the context of the preservation of the non-blocking characteristic of the AmOP paradigm, which is discussed in the next section.

4.5 Discussion

There is one characteristic from the ambient-oriented programming paradigm that needs to be preserved, namely the non-blocking communication operators (section 3.1). We illustrate how the kernel language preserves this characteristic. We rely on two prerequisites: First, the implementation of AmbientTalk does not allow remote communication other than through the mechanisms described in section 4.1. Remember from section 4.4.2 that intervening in the communication process occurs through the use of the mailbox observers. Hence, we cannot introduce other communication constructs. Second, there is the prerequisite that the only data shared among actors are actor references.

It is impossible to construct blocking communication in AmbientTalk. Below is an informal reductio ad absurdum proof for a blocking send operation. Suppose we were able to construct a blocking send operation, which blocks until an acknowledgement message is found in the inbox, via the code below:

```
syncsend(target, msgName, args)::{
  outbox.add(createMessage(thisActor(), target, msgName, args);
  while(not(hasReceivedAcknowledgement(msgName, inbox)),void)
}
```

[4] The identifiers are inbox, outbox, sentbox, rcvbox, required, provided, joined and disjoined.

[5] Such an accept was introduced in CodA [38] to intervene in the object synchronization at the meta-level.

The code places the message that needs to be sent in the `outbox`. Subsequently, the while-code continuously checks the `inbox` to see if an acknowledgement has arrived via the function `hasReceivedAcknowledgement`. At first sight, the code seems to implement a blocking send. However, the code above is not correct. It implies that the mailboxes can be changed by an entity other than the actor during the execution of a method. This conflicts with the operational semantics of the ambient actor model, because it violates the serial mailbox access property we explained in section 4.1.3. Hence, the actor will never receive the acknowledgement while it is iterating over the `inbox` and therefore the blocking send cannot be realized.

Other blocking send operations would have to be constructed in a similar way, because communication based on mailboxes is the only communication featured by AmbientTalk. Therefore other blocking communication constructs would also break the serial mailbox access property. Hence, it is impossible to construct blocking communication in the language.

5. LANGUAGE FEATURES FOR THE AMBIENT PARADIGM

In order to show its practicability we extend AmbientTalk with a number of language constructs originating from different distributed languages and middleware that have proven their merits. The customized reflective operators we will present in this section are redefined in the root object so that they are available to all actors as explained in section 4.4.

5.1 Futures

Asynchronous message passing in the actor model necessitates extensive use of callback methods to process the result of sent messages. This is illustrated by the example introduced in section 4.3: when the method `get` is invoked, a callback `result` message is sent along to return the result to the caller. The use of callbacks complicates the structure of the software a lot and it is a source for race conditions [7].

To resolve these problems we can enrich the language with futures [4, 25, 34] (also called promises). Futures are placeholders for the eventual result of an asynchronous call. They are a proxy for the result to be computed in the future. Once the result is computed the future is said to be resolved and it forwards the messages to the result. With futures programs can be written that use asynchronous communication but that still resemble the control flow of synchronous communication:

```
result: aQueue<-pop();
...
result<-print()
```

In the past, many languages based on futures have been introduced [32, 44, 46, 39, 16]. In these languages the semantics of futures varies. For instance, what happens if the `print` message is sent to the future represented by the `result` variable before that future has been resolved. In some languages this expression blocks until the future has been resolved with the result of the `pop`. Other languages provide a construct to explicitly wait for the result to be computed. These future semantics are in conflict with the *non-blocking communication primitives* characteristic of ambient-oriented programming languages that we advocated in section 3. Hence, although one can write what appears to be sequential programs, they are in fact a hidden source for (distributed) deadlocks.

The concept of futures we integrated in the language is based on the contemporary distributed language E [39]. This type of futures was explicitly designed to support non-blocking communication. Its implementation is given below.

```
future::object({
  resolved: void;

  init()::{
    inbox.addListener(thisActor()<-in)
  };

  resolve(content)::{
    resolved:=content;
    inbox.iterate({
      msg: el;
      inbox.delete(msg);
      forward(msg)
    })
  };

  forward(msg):
    if(is_actor(resolved),
       outbox.add(setMsgTarget(copyMsg(msg),
                               resolved)));

  in(msg)::{
    if(not(is_void(resolved)), {
      inbox.delete(msg);
      forward(msg)
    })
  }
});
```

An execution trace of the queue example from above with non-blocking futures is shown in figure 2. A future is created before a message is sent and is *attached* to the message that is sent. After that message is executed by the target actor a `resolve` message is sent to the future with the result of the execution as its argument. The future object has a `resolve` method that iterates[6] over the `inbox` and forwards its messages to the result. The listener method `in` forwards incoming messages to the result after the future was resolved. The future is subsequently integrated in the message sending process of the language by adapting the metaobject protocol:

```
send(target, name, args)::{
  aFuture: actor(future.new());
  outbox.add(createMessage(thisActor(), target, name, args,
                      aFuture));
  aFuture
};
```

[6]The `iterate` method is defined with a call-by-name argument (discussed in section 4.2) that is parameterized with `el`. The expression provided as an argument is executed by the `iterate` method for each element in the mailbox, with `el` bound to that element.

With each message sent, a new future actor `aFuture` is created. This future actor is attached to the message that has to be sent and is placed in the `outbox`. Finally, this future actor is returned. The `process` method is redefined so that the method associated with the message is invoked and its return value is used to send the `resolve` message to the future actor.

```
process(message)::{
  value: super().process(message);
  aFuture: getFuture(message);
  aFuture<-resolve(value));
  value
};
```

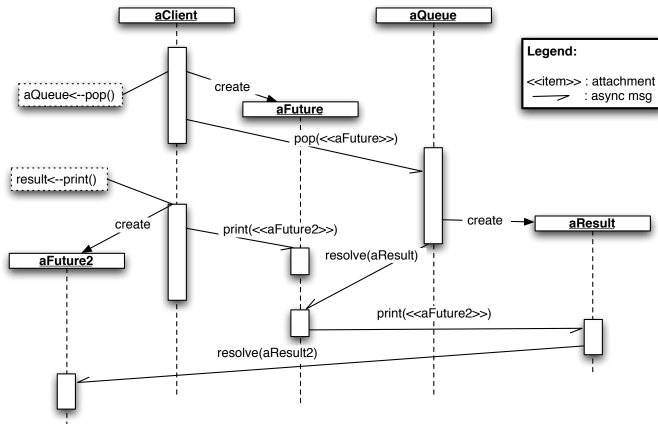

Figure 2: Behavior of Non-Blocking Futures

Sometimes we need to express that certain code is executed only when the future has been resolved. For example, suppose we want to write the following:

```
aFuture: aQueue<-pop();
...
if(aFuture = 1, doSomething())
```

The conditional expression can only be evaluated when `aFuture` has been resolved. To support such expressions we used reflection to integrate a language construct based on the *when-catch* construct that is available in the language E [39]. This construct allows one to specify what code should be executed after the given future has been resolved. Note that the when construct does not block, it will execute its code block asynchronously when the future is resolved. With the `when` construct the above code can be written as follows:

```
aFuture: aQueue<-pop();
...
when(aFuture, if(result = 1, doSomething()))
```

The `when` construct is implemented by adding the observer pattern to the future object we introduced above. Observers are notified when the future receives a `resolve` message[7].

[7]The changes we made to integrate the observer pattern in the future object are standard and therefore we did not list the changed code.

Below is the code for the prototype observer of the future object that is used to support the when construct:

```
futureObserver: object({
  code: void;
  new(customCode)::{ copy(code:=customCode) };
  notify(result)::{ code(result) }
});
```

The `futureObserver` object is initialized with a block of code. Upon notification the block of code is executed with a new variable `result` bound to the resolved future value. The `when` construct is implemented with the following function:

```
when(aFuture, code(result))::{
  aFuture<-subscribe(
      actor(futureObserver.new(code)))
}
```

The first parameter takes a reference to a future actor. The second parameter is a call-by-name argument, as explained in section 4.2 and is used to initialize the `futureObserver` actor. That actor is then used as an argument to add the listener to the future actor.

5.2 Application-Specific Routing

Some applications can benefit from application-specific context information to enhance routing of messages in a mobile ad hoc network [35]. For example, suppose that we have a group of four people A, B, C and D who form an intermittently connected ad hoc network (their mobile devices can only communicate when they are in one another's communication range). The mobile device of A wants to send a message to the mobile device carried by D. Since the expected concrete movements of those people are unknown we do not know when and if A will eventually move in communication range of D so that their devices can communicate directly. For example, it is possible that A and D only meet each other each couple of months, while B and C often meet with either A or D. Hence, we can increase chances of communication between the mobile devices of A and D by relying on the fact that A moves frequently in the communication range of B or C and B or C frequently moves in the communication range of D. Thus, the mobile devices of B and C can be used as a middle man. It is clear that such information can only be deduced from application-specific context information, such as an agenda application, or an application that keeps track of human social networks.

A construct `via` can be added to the language to construct a reference that performs such application-specific routing. The example above can be written as:

```
via(targetOnD, [routerOnB, routerOnC])<-msg()
```

When the device of A sends this message to the device of D it will send the message not only to D, but also to B and C. The router actors must be explicitly provided in the via-construct, because they rely on the social network of the user. In this example `routerOnB` and `routerOnC` run

on mobile devices carried by persons B and C respectively. The construct `via` is supported by the object `routedRef` that implements the redundant message passing mechanism:

```
via(target, routers)::{
  actor(routedRef.new(target, routers))
};

routedRef::object({
  target : void;
  routers: void;
  new(aTarget, routersTbl)::{
    copy({ target:=aTarget; routers:=vector.new(routersTbl) })
  };
  init()::{
    inbox.addListener(thisActor()<-in)
  };
  in(msg)::{
    routers.iterate(el.route(msg));
    outbox.add(msg)
  }
});
```

The object `routedRef` has a method `in` that acts as a mailbox listener on the `inbox`. When a message is added to the `inbox` that message is passed to the `router` actors[8] running on the mobile devices of B and C. When the mobile device of B or C enter the communication range of the mobile device carried by D then the message is communicated by the `router` actor.

Since the message can possibly arrive at its destination multiple times, the receiver has to filter these messages to ensure that the message is processed at most once. This is achieved by overriding the `process` method to check the `rcvbox` to see if the message has already been processed by the actor. If this is the case, it is removed from the `inbox` otherwise the message is processed.

```
process(msg)::{
  if(rcvbox.contains(msg)),
     inbox.remove(msg),
     super().process(msg))
}
```

5.3 Group Communication

In a peer-to-peer environment a peer often needs to communicate directly with multiple peers, because interactions often occur with multiple small objects rather than one object that acts as a server, for instance when a peer-to-peer meeting scheduler wants to query multiple agendas to know if they have a free time-slot. In most current distributed languages a loop must be used to run over the set of remote objects we need to communicate with in order to achieve group communication. When group communication is frequently used these loops start to clutter the code. We extended the language with a set of references based on the many-to-many invocations (M2MI) library [31].

M2MI provides two types of handles that do group communication: first, there are multihandles to express one-to-many communication. The handle is created via the enumeration of the set of references that are involved in the group communication. Sending a message to the multihandle results in a message sent to all the references that were enumerated. Second, there are omnihandles that are used to express many-to-many communication, which is communication with all objects in the communication range that implement a certain interface.

[8]The code of the router actor does nothing more than place the incoming messages in its `outbox` and is therefore not listed.

Both multihandles and omnihandles can be expressed using the reflective operators, but we focus on the latter in this example. The construct `omnihandle` allows many-to-many communication based on the pattern-based lookup mechanism from the AAM, discussed in section 4.1.2. The example below shows how a reservation request could be sent to the agendas of members of the programming technology laboratory and the software engineering laboratory, that enter the communication range of the sending device.

```
agendas: omnihandle("agenda:proglab", "agenda:selab");
agendas<-reserve(time.tomorrow().at(17), "Inter-lab meeting")
```

The omnihandle construct is implemented as an `omnihandleRef` object:

```
omnihandleRef::object({
  patterns: void;
  new(aPatternTbl)::copy({patterns:=aPatternTbl});

  init()::{
    required.addAll(patterns);
    joined.addListener(thisActor()<-joined)
  };

  disable()::{ required.deleteAll(patterns) };

  joined(aResolution)::
    inbox.iterate({
      outbox.add(setMsgTarget(copyMsg(el), provider(aResolution)))
    })
});

omnihandle@patterns::actor(omnihandleRef.new(patterns))
```

The function `omnihandle` takes a variable number of actual arguments (denoted using the @-syntax) to be bound in a table called `patterns`. The `omnihandleRef` actor is initialized by putting that table with patterns in the mailbox `required`. The `omnihandleRef` object has a listener method `joined` on the mailbox with the same name. Each time an actor is in the communication range of another actor that provides the required patterns, the messages from the `inbox` of the omnihandle actor are forwarded to the provider.

5.4 Discussion

As discussed in section 5.1, the low-level communication constructs in the AAM heavily rely on call backs which renders applications written in it very complex.

In this section, we have put AmbientTalks reflection operators discussed in section 4.4 (namely, overriding send and process and the explicit manipulation of mailboxes based on listeners) to work by showing how they can be used to implement a number of high-level communication constructs incorporated in other contemporary distributed languages and middleware.

We are not claiming that these language constructs are the best choice for all applications for mobile networks. It was rather our intention to show how AmbientTalk can be used as a language lab for experimenting with language constructs designed for mobile (ad hoc) networks.

6. CONCLUSION AND FUTURE WORK

In this paper we have defined the ambient-oriented programming (AmOP) paradigm, a set of characteristics for programming languages, which address the hardware phenomena encountered when developing applications for mobile computing. The current state of the art of neither distributed languages nor middleware addresses all the consequences of these hardware phenomena.

We introduced a kernel language named AmbientTalk, adhering to the characteristics of the AmOP paradigm. This kernel can be extended with language features that address coordination and communication issues in mobile computing. We implemented a number of language features to illustrate the extensibility of the kernel language. Our goal was not to show that these language features are the best choice for all applications for mobile computing. It was rather our intention to show how AmbientTalk can serve as a language lab for experimenting with AmOP language features. Future work encompasses additional experimentation with advanced language features. A brief description of each follows:

- A necessary condition of the proposed solution is that the metaobject protocol of all interacting actors must contain the code for their communication abstractions. We are planning experiments with a mobile meta object protocol, where the needed adaptation is attached to the message.
- We are looking into language constructs to deal with inconsistencies in partitioned mobile network environments. One way of resolving such inconsistencies is through the use of rollback mechanisms. We expect that the system of mailboxes are appropriate to model them, because of the explicit access to the computational and communicational state of an actor via the reified communication traces.
- There is a need for new distributed garbage collection algorithms, because most current algorithms are not intended for use in partially connected networks. Moreover, it is possible that part of the network never becomes available again. For example, when a user never returns to a certain place. We believe that, in some cases, language constructs will need to be provided to guide the garbage collector in cleaning up lost network references.

It is clear that a lot of the territory pertaining to language feature design for AmOP remains uncovered. With this paper we hope to promote a new branch of distributed languages, which address the problems of mobile computing from the ground up.

Acknowledgements

Thanks to Kris Gybels and Sebastián González for proofreading this paper and providing us with useful comments.

7. REFERENCES

[1] Agha, G. *Actors—A Model of Concurrent Computation for Distributed Systems*. MIT Press, 1986.

[2] Agha, G., and Hewitt, C. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds., Computer Systems Series. The MIT Press: Cambridge, MA, USA, 1988, pp. 37–53.

[3] Agha, G., Mason, I. A., Smith, S. F., and Talcott, C. L. A foundation for actor computation. *Journal of Functional Programming 7*, 1 (1997), 1–72.

[4] Baker Jr., H. G., and Hewitt, C. The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages* (1977), vol. 8 of *ACM Sigplan Notices*, pp. 55–59.

[5] Bentley, J. Programming pearls: little languages. *Commun. ACM 29*, 8 (1986), 711–721.

[6] Briot, J.-P., Guerraoui, R., and Löhr, K.-P. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys 30*, 3 (September 1998), 291–329.

[7] Briot, J.-P., and Yonezawa, A. Inheritance and synchronization in concurrent oop. In *European conference on object-oriented programming on ECOOP '87* (1987), Springer-Verlag, pp. 32–40.

[8] Callsen, C. J., and Agha, G. Open heterogeneous computing in actorspace. *J. Parallel Distrib. Comput. 21*, 3 (1994), 289–300.

[9] Caporuscio, M., Carzaniga, A., and Wolf, A. L. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. Software Engineering 29*, 12 (dec 2003), 1059–1071.

[10] Cardelli, L. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* (New York, NY, 1995), pp. 286–297.

[11] Cardelli, L., and Gordon, A. D. Mobile ambients. In *Electronic Notes in Theoretical Computer Science* (2000), A. Gordon, A. Pitts, and C. Talcott, Eds., vol. 10, Elsevier.

[12] Chiba, S., and Masuda, T. Designing an extensible distributed language with a meta-level architecture. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming* (1993), Springer-Verlag, pp. 482–501.

[13] Cugola, G., and Jacobsen, H.-A. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev. 6*, 4 (2002), 25–33.

[14] Cugola, G., Nitto, E. D., and Pico, G. P. Content-based dispatching in a mobile environment. In *Proceedings of WSDAAL* (2000), ACM Press.

[15] Davies, N., Friday, A., Wade, S. P., and Blair, G. S. An asynchronous distributed systems platform for heterogeneous environments. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications* (1998), ACM Press, pp. 66–73.

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

[17] De Meuter, W., D'Hondt, T., and Dedecker, J. Intersecting classes and prototypes. In *Ershov Memorial Conference* (2003), M. Broy and A. V. Zamulin, Eds., vol. 2890 of *Lecture Notes in Computer Science*, Springer. Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003.

[18] De Meuter, W., D'Hondt, T., and Dedecker, J. Pico: Scheme for mere mortals. In *1st European Lisp and Scheme Workshop, Oslo, Norway* (2004).

[19] Dedecker, J., and Van Belle, W. Actors for mobile ad-hoc networks. In *Embedded and Ubiquitous Computing* (2004), L. Yang, M. Guo, G. Gao, and N. Jha, Eds., vol. 3207 of *Lecture Notes in Computer Science*, Springer, pp. 482–494. Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004.

[20] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2 (2003), 114–131.

[21] Fjellheim, T., Milliner, S., Dumas, M., and Elms, K. The 3dma middleware for mobile applications. In *Embedded and Ubiquitous Computing* (2004), L. Yang, M. Guo, G. Gao, and N. Jha, Eds., vol. 3207 of *Lecture Notes in Computer Science*, Springer, pp. 312–323. Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004.

[22] Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Jan. 1985), 80–112.

[23] Gong, L. JXTA for J2ME extending the reach of wireless with JXTA technology. Tech. rep., SUN Microsystems, http://www.jxta.org/project/www/docs/JXTA4J2ME.pdf, 2002.

[24] Haahr, M., Cunningham, R., and Cahill, V. Supporting CORBA applications in a mobile environment. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom-99)* (N.Y., Aug. 15–20 1999), ACM Press, pp. 36–47.

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

[26] Hewitt, C. E. Viewing control structures as pattern of passing messages. *Artificial Intelligence: An International Journal 8*, 3 (June 1977), 323–364.

[27] ISTAG. Ambient intelligence: from vision to reality, September 2003. Draft report.

[28] Joseph, A., Tauber, J., and Kaashoek, F. Mobile computing with the rover toolkit. *IEEE Transactions on Computers 46*, 3 (Mar. 1997), 337–352.

[29] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109–133.

[30] Kahn, K. M., and Saraswat, V. A. Actors as a special case of concurrent constraint programming. In *Proc. of the OOPSLA/ECOOP-90: Conference on Object-Oriented Programming: Systems* (Languages, and Applications / European Conference on Object-Oriented Programming, Ottawa, Canada, 1990), pp. 57–66.

[31] Kaminsky, A., and Bischof, H.-P. Many-to-many invocation: A new object oriented paradigm for ad hoc collaborative systems. *17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)* (2002).

[32] Karaorman, M., and Bruno, J. A concurrency mechanism for sequential eiffel. In *Proceedings of the eighth international conference on Technology of object oriented languages and systems* (1992), Prentice-Hall, Inc., pp. 63–77.

[33] Liskov, B. Distributed programming in argus. In *Distributed Computing Systems: Concepts and Structures*, A. L. Ananda and B. Srinivasan, Eds. IEEE Computer Society Press, Los Alamos, CA, 1992, pp. 370–382.

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

[35] Mamei, M., and Zambonelli, F. Programming pervasive and mobile computing applications with tota middleware. In *Embedded and Ubiquitous Computing* (2004), IEEE, pp. 263–??? Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04), Orlando (FL), U.S.A., March, 2004.

[36] Mascolo, C., Capra, L., and Emmerich, W. Mobile computing middleware. In *Advanced lectures on networking*, vol. 2497. Springer-Verlag New York, Inc., 2002, pp. 20–58.

[37] Masuhara, H., Matsuoka, S., and Yonezawa, A. Implementing parallel language constructs using a reflective objectoriented language. In *Proceedings of Reflection Symposium'96* (Apr. 1996), pp. 79–91.

[38] McAffer, J. Meta level programming with CodA. In *Proceedings of ECOOP'95, Aarhus, Denmark*, W. Olthoff, Ed., Lecture Notes in Computer Science 952. Springer-Verlag, Berlin, 1995, pp. 190–214.

[39] Miller, M. The E programming language, the secure distributed pure-object platform and p2p scripting language for writing capability-based smart contracts. http://www.erights.org.

[40] Murphy, A. L., Picco, G. P., and Roman, G.-C. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems* (2001), IEEE Computer Society, p. 524.

[41] Reynolds, P., and Brangeon, R. DOLMEN - service machine development for an open long-term mobile and fixed network environment, Feb. 19 1999.

[42] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4 (1990), 447–459.

[43] Schill, A., Bellmann, B., Bohmak, W., and Kummel, S. System support for mobile distributed applications. In *SDNE '95: Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments* (1995), IEEE Computer Society, p. 124.

[44] Taura, K., Matsuoka, S., and Yonezawa, A. ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of DIMACS '94 Workshop* (1994), G. E. Blelloch, K. M. Chandy, and S. Jagannathan, Eds., vol. 18. Specification of Parallel Algorithms of *Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, pp. 275–291.

[45] Terry, D. B., Petersen, K., Spreitzer, M. J., and Theimer, M. M. The case for non-transparent replication: Examples from Bayou. *IEEE Data Engineering Bulletin 21*, 4 (dec 1998), 12–20.

[46] Varela, C., and Agha, G. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices 36*, 12 (2001), 20–34.

[47] Varela, C. A., and Agha, G. A. What after java? from objects to actors. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7* (1998), Elsevier Science Publishers B. V., pp. 573–577.

[48] Weiser, M. The computer for the 21st century. *Scientific American 265*, 3 (1991), 66–75.

[49] Zachariadis, S., Capra, L., Mascolo, C., and Emmerich, W. XMIDDLE: information sharing middleware for a mobile environment. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)* (New York, May 19–25 2002), ACM Press, pp. 712–712.