Abstractions for Context-aware Object References

Tom Van Cutsem* Jessie Dedecker* Stijn Mostinckx† Wolfgang De Meuter
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium
{tvcutsem,jededeck,smostinc,wdmeuter}@vub.ac.be

ABSTRACT

Within the domain of pervasive, mobile computing, we consider context-awareness in terms of the set of available services in direct proximity to a mobile device. We seek new programming language constructs to represent and influence such contextual information more expressively. One important aspect is the discovery and addressing of available, proximate, services. To this end, we investigate abstractions based on object references, which we term *ambient references*: references which point into the hardware environment surrounding the program.

1. INTRODUCTION

We focus on ubiquitous computing in the form of pervasive wireless networks composed of mobile devices. Today, these devices are PDAs or cell phones, but in the future such pervasive wireless networks may include smart wristwatches or intelligent television screens, as e.g. envisioned by Weiser [7]. Applications running on devices roaming in such hardware environment must clearly be context-aware: devices move about constantly and in doing so directly influence their surrounding context.

In this paper, we consider only a small subset of potential context information, to wit the physical location of devices. This physical location directly relates to the proximate set of available services offered by other devices in the environment. Such a "proximate set" is dynamically demarcated by the limited range of the wireless network connections. We consider each device to host an object or component system, where some objects or components are made available to peer devices as services.

Our goal is to build programming language abstractions for *addressing* the services located on remote devices. As a concrete example, consider a printer with a built-in printing service program. When the user declares that he wants to print a file from his PDA, and he is in close proximity to the printer, an appropriate service discovery algorithm should bring the PDA's objects in contact with the printing service. In regular object systems, acquaintance relations between objects are represented as object references (i.e. "pointers"). We therefore seek to explore abstractions for object references which can denote remote objects on a context-sensitive basis. The goal of such references is to both discover remote objects and to become a reference (i.e. a communication channel) to them. We name such object references *ambient references*.

*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)

2. AMBIENT REFERENCES IN A NUTSHELL

Recall the problem of finding a suitable printing service in the environment in order to print a file from a mobile device. The problem can be decomposed in two tasks: *discovering* a printing service and, if one is successfully found, *communicate* with this printing service. These two aspects are reflected in our notion of an ambient reference: an ambient reference discovers a suitable service automatically *and* becomes a representative of (i.e. a proxy to) the service which can be sent messages. As an example, the printer problem can be solved using an ambient reference in pseudocode as follows:

```
aPrinter = ambient Printer;
aPrinter.print(aDocument);
```

The expression `ambient Printer` literally means "find me an object in the ambient offering a Printer service". Messages can be sent to the ambient reference even if a suitable printer has not yet been discovered. We will go into more detail on how services are denoted in the following section.

3. ASPECTS OF DISCOVERY AND COMMUNICATION

We consider two aspects of ambient references: how they denote remote objects and how messages sent to those objects are evaluated.

Discovery

It is important for objects to denote other objects on a loosely-coupled, device-independent, intensional basis, rather than through the use of a fixed global address or URI. As the environment surrounding a mobile device is changing constantly, and services appear and disappear unheraldedly, it becomes infeasible to know the exact address of a service upfront. For the purposes of this paper, we consider an intensional description of an object to be its

interface (which need not necessarily be aligned with a static type description as in Java). In the printer example shown previously, we consider `Printer` to denote a function selecting only objects implementing a certain `Printer` interface.

Communication

Ambient references can be sent messages just like normal object references. Messages sent via ambient references are handled asynchronously: the sender will not wait for the message to be processed by the receiver. Our reasons for adopting asynchronous, non-blocking message passing are threefold:

- Because remote method invocation is much slower than local method invocation, especially in wireless networks, the asynchrony can be used to overlap execution of sender and receiver, which better hides network latency.
- Asynchronous message passing decouples sender and receiver in time: a message can be sent to a receiver even when it is not online (connected) at the time the message is sent [6]. This is made possible by decoupling message sending from message delivery: messages which cannot be transmitted immediately are stored in an outgoing message queue internal to the ambient reference.
- As a consequence of decoupling message delivery from message sending, a sender object will not have to synchronise with (i.e. wait for) its remote communication partner for every message send, which increases the overall responsiveness of the system. Blocking communication more easily leads to unresponsive (and possibly deadlocked) systems.

An important aspect of distributed communication is *failure*. Note that, because of asynchronous, non-blocking message passing, the unavailability of the receiver is not considered a failure or an exception: the message is kept until the receiver becomes available. This design decision is founded on the observation that disconnections in pervasive networks are commonplace due to the volatile, wireless connections. If necessary, failures can be induced using timeouts. A description of the necessary language constructs to gracefully handle these failures is outside the scope of this paper.

Asynchronous message sends are usually not attributed any return value, which requires the use of "callback" methods in order to process results. Such programming idioms clutter the code, which is why we adopt the use of *futures* or *promises* [3, 8, 5]. An asynchronous message send always immediately returns a future object, which is a placeholder for the real return value. Once the real value is computed, it can be extracted from the future object by the sender.

4. DESIGN DIMENSIONS

We decompose the design space of ambient references along two dimensions: *cardinality* and *propensity*. The cardinality of an ambient reference is the number of objects the reference denotes. We distinguish between mono-ambient references and multi-ambient references. The former resemble classic object references and denote at most one object, the latter resemble object groups or collections and refer to an arbitrary number of objects.

The propensity of an ambient reference can be either weak or strong. The referent(s) of a weak ambient reference can change over time, whereas the referent(s) of a strong ambient reference are fixed. Conceptually, one should think about a strong ambient reference as denoting a well-defined set of objects, whereas weak ambient references denote a cloud of objects, whose boundaries are vague and change constantly. This "cloud" is delimited by the wireless communication range of the mobile device. That is, a weak ambient reference can only refer to objects which are in physical proximity to the device. Hence, the difference between weak and strong ambient references can be characterised by the topology of the objects they denote: a strong ambient reference defines a logical topology between objects regardless of physical properties of the environment, whereas a weak ambient reference defines a physical topology, constantly changing in unison with the environment.

An ambient reference supports operations similar to ordinary object references: it can be sent messages, it can be bound to variables, passed as an argument, etc. In addition, we envision an extra operator, named `snapshot`, which allows the programmer to change the propensity dimension of an ambient reference. The expression `snapshot(weakRef)` returns a strong version of a weak ambient reference. Conceptually, it allows one to take a snapshot of the – continuously changing – environment. The use of this operator is detailed in the following section.

5. A TAXONOMY OF AMBIENT REFERENCES

Our decomposition gives rise to four kinds of ambient references: weak and strong mono-ambient references and weak and strong multi-ambient references. We now explore the design space of ambient references and identify tentative abstractions for each kind of reference in the hardware context of pervasive computing.

5.1 Weak Mono-ambient References

A weak mono-ambient reference denotes at most one object which is proximate to (i.e. in communication range with) the host device. Figure 1 illustrates mono-ambient references. The white object holds an ambient reference which matches a certain interface (denoted by an endpoint with an abstract shape). Objects to the left of the dotted line are in the ambient reference's communication range. The figure on the left describes an initial situation where a weak ambient reference is created which is unbound (shown dangling): there are no proximate objects offering the required service. On the right of the figure, a situation is described where a matching object is within communication range. The ambient reference becomes bound, but remains weak (denoted by the squiggled line): it can rebind to other matching objects and becomes unbound again should the referent object move out of range.

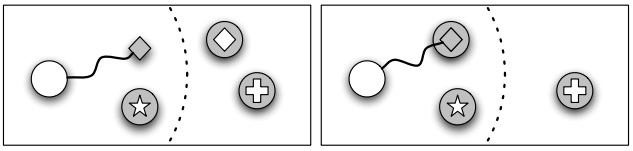

Figure 1: Unbound (left) and Bound (right) Weak Mono-ambient References

Note that messages can be sent to the ambient reference at any time. If the ambient reference is unbound, any messages sent to it are enqueued and delivered when the reference becomes bound.

We consider ambient references to be typically used for discovering such services as “a proximate printer”, “a proximate projection screen”, “a proximate mail server”, etc. The printer example in section 2 illustrated the use of a weak mono-ambient reference.

5.2 Strong Mono-ambient References

Weak references are ideal to discover proximate objects, but they are less interesting for sustaining interactions with such objects. A bound weak reference can become unbound at any point in time and be rebound to another object. As such, subsequent messages sent to a weak reference can be received by different objects, which is not always desirable. Strong references provide a solution to this problem: once a strong reference is bound, it remains bound to the same object, even if this object moves out of range. Figure 2 illustrates strong mono-ambient references.

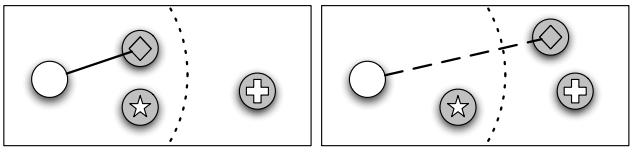

Figure 2: Connected (left) and Disconnected (right) Strong Mono-ambient References

The left-hand side of the figure describes a situation where a strong reference is bound to a proximate object. The interesting difference with weak references is shown on the right-hand side: the reference remains bound even though the object is no longer proximate. The programmer is not obliged to consider the state of connectedness of the strong reference: messages sent to a disconnected reference will be transmitted whenever the object becomes proximate again.

In order to create strong ambient references, the `snapshot` operator is used. Applying `snapshot` to a weak mono-ambient reference creates a strong mono-ambient reference, which is bound to the object the weak ambient-reference is bound to. If the weak reference is unbound at that time, the strong reference will be bound to the first object subsequently bound to the weak reference.

As an example, consider the printing service again. Imagine one were to send not one, but a batch of documents to the printer. If it is desirable that all documents are printed by the same printer, this can be achieved with a strong mono-ambient reference as follows (presuming the variable `batch` denotes a queue of documents):

```
aPrinter = snapshot(ambient Printer);
foreach document in batch {
  aPrinter.print(document);
}
```

5.3 Weak Multi-ambient References

Multi-ambient references are characterised by the fact that they represent sets of objects. These sets are described intensionally (i.e. one does not explicitly list all members of the set). Note that, since an empty set is a valid set, a multi-ambient reference can never really be “unbound”. If the set is empty, messages sent to the reference are lost. The left-hand side of figure 3 depicts a weak multi-ambient reference. The reference consists of a volatile set of all matching proximate objects.

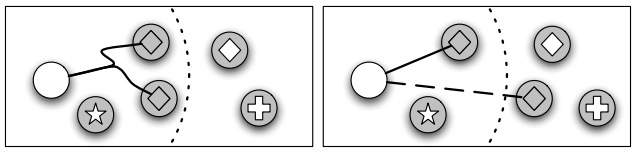

Figure 3: Weak (left) and Strong (right) Multi-ambient References

Weak multi-ambient references are not “collections” of objects in the typical object-oriented sense. No constructs are provided to test for set membership, to add or delete objects or to iterate over the objects referred by the multi-ambient reference. Weak multi-ambient references form an ideal abstraction for “shouting” information to proximate devices. One can imagine exhibits in a museum to “shout” a digital description of themselves to the PDAs of passing visitors. Similarly, base stations in airports or railway stations could broadcast information like departure times and delays to the PDAs of interested travellers. Weak multi-ambient references are directly geared towards expressing such behaviour[1]:

```
travellers = ambient* Traveller;
travellers.announce(delays);
```

5.4 Strong Multi-ambient References

Although weak multi-ambient references are fine for broadcasting and discovery, like their mono-ambient cousins, they are not suitable for stable interactions. The final type of ambient reference covers such multicast interactions. The right-hand side of figure 3 illustrates the major difference with weak multi-ambient references: the set of referents denoted by a strong multi-ambient reference is maintained regardless of the communication range of devices. Whereas the set intensionally described by weak multi-ambient references only consists of proximate objects, this is no longer the case for strong ambient references. As to be expected, a strong multi-ambient reference can be created by taking a snapshot of a weak multi-ambient reference.

In contrast to the weak reference, the strong reference *does* denote a collection of objects. Because the set of objects belonging to the strong reference is well-defined, it makes sense to allow e.g. an enumeration of the referents of a strong multi-ambient reference. A useful idiom is the creation of a weak multi-ambient reference which can be inspected by regularly enumerating a snapshot of the reference. For example, the following code can be used to “list all proximate printers” by invoking the `list` method on an object holding a weak multi-ambient `Printer` reference:

```
printers = ambient* Printer;
method list() {
  foreach printer in snapshot(printers) {
    doSomethingWith(printer);
  }
}
```

Strong multi-ambient references are very similar to classical object group abstractions used to implement chat services, distributed

[1] We use an asterisk to denote a multi-ambient reference.

whiteboards, etc. It is, however, important to recall that messages sent to the group are handled asynchronously and can be sent even when not all group participants are present. As for the return value of such multicast asynchronous message sends, we are looking into "multifutures" which are adaptations of futures able to collect multiple return values.

6. VALIDATION AND RELATED WORK

We are currently experimenting with ambient references in a small prototype programming language called AmbientTalk [2], specifically designed for ubiquitous computing. The language is based on the actor paradigm for concurrent object-oriented programming [1]. We have validated the design of mono-ambient references by building a small instant-messaging application where the different peers discover and communicate with one another using mono-ambient references[2]. The ambient references allow the instant messenger to be deployed in ad hoc networks without any infrastructure, and make peer communication resilient to disconnections.

Our notion of an ambient reference is very similar to that of a *handle* in the many-to-many invocations (M2MI) paradigm [4]. M2MI handles use Java interfaces to denote other objects in a loosely coupled fashion and also employ asynchronous message passing. M2MI distinguishes between *unihandles*, *multihandles* and *omnihandles*. Roughly speaking, unihandles resemble strong mono-ambient references, while multi- and omnihandles resemble strong and weak multi-ambient references respectively. An omnihandle represents all objects in communication range implementing the handle's interface. A message sent to an omnihandle means "every object out there that implements this interface, call this method" [4].

Although M2MI was of great influence to the design of our ambient references, there are some important differences. First, M2MI offers no delivery guarantees: if a message is sent to an object which is not in communication range at that time, the message is lost. Hence, message sending and delivery are not decoupled as is the case with ambient references. The consequence is that the responsibility of guaranteed message delivery is passed on to the application itself. A second difference is that messages sent to M2MI handles do not return a value, requiring the use of callbacks as explained previously. Third, the construction of uni- and multihandles differs from the creation process of strong ambient references. In M2MI, objects must be explicitly attached to a handle, i.e. the set denoted by such a handle is explicitly enumerated. Strong ambient references dynamically "discover" their set content by taking a snapshot of a weak reference. In M2MI, there is no notion of such "snapshots".

[2]Our programming language platform showcasing this instant messenger has been accepted for demonstration at the OOPSLA '05 conference.

7. POSITION STATEMENT

Ubiquitous computing requires programming language constructs that abstract from the complex hardware environment while remaining translucent enough to deal with the inescapable issues of distributed computing. When objects are distributed among mobile devices connected in a volatile network, it is no longer trivial to discover and communicate with remote parties. We advocate that object references ought to be augmented with context information: they should become "aware" of the hardware constellation surrounding their host device. Rather than identifying one such new kind of object reference, we have identified a number of design dimensions whose combinations give rise to different kinds of references, each suitable for a different kind of collaboration.

8. REFERENCES

[1] AGHA, G. Concurrent object-oriented programming. *Communications of the ACM 33*, 9 (1990), 125–141.

[2] DEDECKER, J., VAN CUTSEM, T., MOSTINCKX, S., DE MEUTER, W., AND D'HONDT, T. AmbientTalk: A Small Reflective Kernel for Programming Mobile Network Applications. Tech. rep., Vrije Universiteit Brussel, 2005.

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

[4] KAMINSKY, A., AND BISCHOF, H.-P. 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* (New York, NY, USA, 2002), ACM Press, pp. 72–73.

[5] 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.

[6] MASCOLO, C., CAPRA, L., AND EMMERICH, W. Mobile Computing Middleware. In *Advanced lectures on networking*. Springer-Verlag New York, Inc., 2002, pp. 20–58.

[7] WEISER, M. The computer for the twenty-first century. *Scientific American* (september 1991), 94–100.

[8] YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications* (1986), ACM Press, pp. 258–268.