Programming mobile peer-to-peer applications with AmbientTalk

Technical Report VUB-SOFT-TR-13-05

Tom Van Cutsem*[1], Elisa Gonzalez Boix†[1], Christophe Scholliers‡[1], Andoni Lombide Carreton[1], Dries Harnie†[1], Kevin Pinte[1] and Wolfgang De Meuter[1]

[1]Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels, Belgium

Abstract

The rise of mobile computing platforms has given rise to a new class of applications: mobile applications that interact with peer applications running on neighbouring phones. Developing such applications using current technology is a challenge because of problems inherent to concurrent and distributed programming, and because of problems inherent to mobile networks, such as the fact that wireless network connections are generally less stable.

We present AmbientTalk, a distributed programming language designed specifically to develop mobile peer-to-peer applications. We discuss the language's foundations and our experiences in using it. We focus in particular on the language's concurrency and distribution model since it lies at the heart of AmbientTalk's support for responsive, robust application development. The model is based on communicating event loops, itself a flavour of the actor model. We provide a precise description of this model by means of a small-step operational semantics. To the best of our knowledge, this is the first formal coverage of an actor language based on communicating event loops.

Keywords: Event loops, actors, futures, mobile networks, peer-to-peer

1 Introduction

Throughout the past decade, we have seen the rise of mobile platforms such as J2ME, iOS and Android. These platforms, in turn, enable a new class of applications: *mobile peer-to-peer (P2P) applications*. What is characteristic of such applications is that they are often used on the move, and that they sporadically interact with peer applications running on neighbouring phones (often communicating via a wireless ad hoc network [MLE02]).

*Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

†Funded by the Prospective Research for Brussels (PRFB) program of the Brussels Institute for Research and Innovation (Innoviris).

‡Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

Developing such applications is a challenge not only because of the inherent difficulty of developing distributed applications. Connectivity between phones is often intermittent (connections drop and are restored as people move about) and applications may not always rely on fixed infrastructure or a reachable central server to support the coordination.

In this paper we present AmbientTalk, a distributed programming language designed specifically to develop mobile P2P applications. To the best of our knowledge, AmbientTalk is the first distributed object-oriented language that specifically targets applications deployed on mobile phones interconnected via an ad hoc wireless network. On the surface, the language is similar to other OO scripting languages such as Javascript, Ruby or Python. However, contrary to these languages, it offers built-in support for concurrent and distributed programming. Its concurrency model is founded on actors [Agh86]: loosely coupled, asynchronously communicating components.

To support distributed programming, AmbientTalk has built-in support for service discovery (built on top of UDP), remote messaging (built on top of TCP/IP), failure handling, asynchronous event processing and publish/subscribe coordination between distributed applications. The current AmbientTalk implementation is an interpreter, written in Java, and specifically targets Android-powered smartphones. It is open sourced under an MIT license and available at `ambienttalk.googlecode.com`.

This paper presents the most complete description of AmbientTalk to date. It complements previous expositions [VMG+07, DVM+06] with an operational semantics of its key features, and provides an overview of the example applications and language extensions developed with and for the language over the past 6 years.

2 Mobile ad hoc networks

AmbientTalk's concurrency and distribution features are tailored specifically to mobile ad hoc networks. We briefly describe the features characteristic of mobile ad hoc networks and why they present a challenge.

There are two discriminating properties of mobile networks, which clearly set them apart from traditional, fixed computer networks: applications are deployed on *mobile* devices connected by *wireless* communication links with a limited communication range. Such networks exhibit two phenomena which are rare in their fixed counterparts:

- **Volatile Connections.** Mobile phones equipped with wireless media possess only a limited communication range, such that two communicating phones may move out of earshot unannounced. The resulting disconnections are not always permanent: the phones may meet again, requiring their connection to be re-established. Often, such *transient* network partitions should not affect an application, allowing it to continue its collaboration transparently upon reconnection. Partial failure handling is not a new ingredient of distributed systems, but these more frequent transient disconnections do expose applications to a much higher rate of partial failure than that which most distributed languages or middleware have been designed for. In mobile networks, disconnections become so omnipresent that they should be considered the rule, rather than an exceptional case.

- **Zero Infrastructure.** In a mobile network, phones (and thus the applications they host) may spontaneously join or leave the network. Moreover, a mobile ad hoc network is often not administered by a single party. As a result, in contrast to

stationary networks where applications usually know where to find collaborating services via URLs or similar designators, applications in mobile networks have to *discover* partner applications while roaming. Services must be discovered on proximate phones, possibly without the help of shared infrastructure. This lack of infrastructure requires a *peer-to-peer* communication model, where services can be directly advertised to and discovered on proximate phones.

Any application designed for mobile networks has to deal with these phenomena. It is therefore worth investigating models or frameworks that ease the development of mobile P2P applications. Because the effects engendered by partial failures or the absence of remote services often pervade the entire application, it is difficult to apply traditional library or framework abstractions. Therefore, support for distributed programming is often dealt with in dedicated middleware or programming languages. This is the main motivation for the design of AmbientTalk as a new programming language.

3 Standing on the shoulders of giants

We briefly describe the foundations of AmbientTalk: where did its features originate?

Object Model AmbientTalk is a dynamically typed, object-oriented language. It was heavily inspired by Self [US87] and Smalltalk [GR89]. Like Ruby, however, AmbientTalk is text-based (not image-based). Inspired by Scheme [SJ75] and E [MTS05], it places an additional emphasis on lexical nesting of objects and lexical scoping.

Concurrency AmbientTalk embraces actor-based concurrency [Agh86]. In particular, it embraces a particular flavor of actor-based concurrency known as *communicating event loops*, pioneered by the E programming language [MTS05], whose distinguishing features are (a) the treatment of an actor as a coarse-grained component that contains potentially many regular objects, and (b) the complete absence of blocking synchronization primitives. All interaction among actors is purely asynchronous.

The event loop model maps well onto the inherently event-driven nature of mobile P2P applications. Phones may join or leave the network and messages can be received from remote applications at any point in time. All of these events are represented as messages sent to objects, orderly processed by actors. The use of event loops avoids low-level data races that are inherent in the shared-memory multithreading paradigm [Ous96, Lee06].

Remote Messaging AmbientTalk avoids traditional RPC-style synchronous distributed interactions, and provides only asynchronous message passing. This was a deliberate design choice to deal with the latency of wireless connections and the intermittent connectivity of devices due to transient network partitions.

Inspired by the queued RPC mechanism of the Rover toolkit [JdT+95], remote references in AmbientTalk automatically buffer outgoing messages for which the recipient is currently unavailable. This allows the communication subsystem to automatically mask temporary network failures, which is especially useful in the face of intermittent wireless connectivity.

AmbientTalk uses leasing to deal with partial failures, inspired by Jini [Wal01].

Following ABCL [YBS86], Eiffel// [Car93], E [MTS05] and Argus [LS88], AmbientTalk features futures (aka promises) to enable return values for asynchronous method calls. This mitigates part of the inversion of control that is characteristic of asynchronous, event-driven code.

Discovery AmbientTalk makes use of the publish/subscribe paradigm [EFGK03] to express discovery among objects: services publish themselves in the network, while clients subscribe to these service announcements. In this light, AmbientTalk is a close cousin of Jini [Wal01], albeit tailored to peer-to-peer networks: AmbientTalk programs need not rely on third-party lookup service infrastructure, but can discover one another directly.

AmbientTalk was also inspired by M2MI [KB02], a lightweight extension to Java enabling asynchronous anycast communication in wireless networks.

Reflection AmbientTalk is meant to serve as a research language to explore the language design space for mobile P2P applications. To support this role, it features an extensive set of reflective APIs to be able to extend the language from within itself. AmbientTalk supports a reflective architecture based on mirrors [BU04] and a variety of hooks into the actor system's message processing and transmission protocols, inspired by early work on reflection in concurrent object-oriented languages [WY88, McA95].

4 Sequential AmbientTalk

Before explaining the concurrent and distributed features of AmbientTalk, we give a brief overview of its more conventional sequential building blocks.

Objects AmbientTalk is a dynamically typed, object-oriented language. It is prototype-based rather than class-based, meaning that objects are not instantiated from class declarations, but rather can be created as anonymous singleton objects (using an object literal declaration) or by cloning existing objects.

In the example below, a top-level function named `makePoint` is defined. Its return value is a fresh object with three slots: `x`, `y` and `distanceToOrigin`. The `x` and `y` slots are initialized with the arguments to the function. The `distanceToOrigin` slot contains a method. Methods are implicitly parameterized with a `self` pseudo-variable, which they can use to access the receiver object's slots. Note that `x` and `y` are instance variables (slots) of the point object, while the variables `t1` and `t2` are local variables of the `distanceToOrigin` method. Numbers are objects in AmbientTalk, and `sqrt` is a method defined on such number objects.

```
def makePoint(x0, y0) {
  object: {
    def x := x0;
    def y := y0;
    def distanceToOrigin() {
      def t1 := self.x * self.x;
      def t2 := self.y * self.y;
      (t1 + t2).sqrt()
    }
  }
};
def p := makePoint(1,1);
p.x; // 1
p.distanceToOrigin(); // 1.4142135623730951
```

Blocks While AmbientTalk is predominantly object-oriented, it has a distinctly functional flavor, through the use of blocks. Blocks (the terminology stems from Smalltalk) are objects that represent anonymous closures, i.e. functions that may refer to lexically enclosing variables. Blocks are constructed by means of the syntax `{|args| body}`, where the `|args|` part can be omitted if the block takes no arguments. For example:

```
def sum := { |x,y| x + y }; // define a block
sum(1,2) // 3
```

Like Smalltalk and Self, AmbientTalk often uses blocks to represent *delayed* computations, such as the branches of an `if:then:else:` control structure, or as listeners or callbacks to await an event, as will be shown later. For instance:

```
def abs(x) {
  if: (x < 0) then: { -x } else: { x }
}
```

The `abs` function calculates the absolute value of a number `x`. The `if`-test is not a built-in statement. Instead, the body of this function consists of a call to the function `if:then:else:`, which expects a boolean and two blocks. If the boolean is true, the first block is called (with no arguments), otherwise the second block is called.

A unique feature of AmbientTalk is that functions or methods can be defined or called using both traditional C-style syntax as well as Smalltalk-style keyword message syntax. In general, keyword message syntax is used to express control structures (such as the `if:then:else:` function) while the C-style syntax is used to express application-level functions or methods (e.g. a function call like `sum(1,2)` or method invocation like `p.distanceToOrigin()`).

Blocks are often used as arguments to higher-order functions, e.g. to map a function over an array. In the following example, a block is mapped over an array of points, producing an array of only the x-coordinates:

```
def xcoords := points.map: { |p| p.x }
```

The keyword message syntax in combination with syntactically lightweight blocks enables AmbientTalk programmers to easily define their own control structures. We have found this to be extremely helpful in a language that makes heavy use of asynchronous APIs and an event-driven programming style.

Type tags Since AmbientTalk is dynamically typed, it cannot use a static type system to categorize objects. Instead, the language provides annotations called type tags. Type tags can be used to annotate whole objects or individual methods or messages. They are also used for service discovery, as described shortly. An example:

```
deftype Fruit;
deftype Apple <: Fruit;
def a := object: {} taggedAs: [Apple];
is: a taggedAs: Fruit // true
```

`Apple` is defined as a subtype of the `Fruit` type tag. The empty object `a` is then annotated with this tag. Type tags are analogous to empty "marker" interfaces in Java (such as `java.lang.Cloneable` and `java.io.Serializable`): these interfaces serve no purpose other than to classify objects, without making any claims as to the objects' supported methods (since these interfaces are empty).

JVM Interoperability AmbientTalk provides built-in support to interoperate with the underlying JVM. This interoperability is similar to that of other dynamic languages implemented on top of the JVM such as Groovy, Jython and JRuby. Concretely, AmbientTalk programs can access Java classes or objects as if they were AmbientTalk objects. This allows AmbientTalk programs to reuse Java libraries. For example, the Java AWT GUI library can be used from AmbientTalk as follows:

```
def b := java.awt.Button.new("Click me");
b.addActionListener(object: {
  def actionPerformed(actionEvent) {
    system.println("The button was pressed");
  }
});
```

This code creates an AWT Button and registers a callback object to be notified when the button is clicked. Contrary to most JVM scripting languages, our interoperability layer takes special care to uphold AmbientTalk's actor-based concurrency model. Concretely, in the above example, when the Java GUI thread invokes `actionPerformed` on the AmbientTalk callback object, the interoperability layer will convert this method call into a message, and post this message to the AmbientTalk actor's event queue, such that the method body will be executed by the actor, not by the Java thread. This avoids race conditions on the object's state, since otherwise both an AmbientTalk actor and a Java thread might concurrently modify it. Our interoperability mechanism is described in full detail elsewhere [VMD09].

Other features The above only scratches the surface of AmbientTalk's features. Two other features worth mentioning are:

- Reflection. AmbientTalk features an extensive reflection API based on mirrors [MVCT+09]. This allows objects to be inspected and modified at runtime. AmbientTalk also supports reflection at the actor-level, allowing for instance access on an actor's incoming message queue.
- Object composition. AmbientTalk features inheritance among objects (as in Self). It also supports traits [SDNB03], a more robust alternative to multiple inheritance. AmbientTalk traits are described in full detail elsewhere [VCBDM09].

5 Concurrent AmbientTalk

As mentioned previously, AmbientTalk's concurrency model is based on actors [Agh86]. A single AmbientTalk virtual machine can host multiple actors that may run in parallel.

5.1 Communicating Event Loops

AmbientTalk combines objects with actors based on the communicating event loops model of the E programming language [MTS05]. What sets this model apart from most other actor languages (such as Act1 [Lie87], ABCL [YBS86], Actalk [Bri88], Salsa [VA01], Erlang [AVWW96], Kilim [SM08], ProActive [BBC+06] or Scala actors [HO07]), is that:

- Each actor is not itself represented as a single object (a so-called "active object"), but rather as a *vat* containing an entire heap of regular objects. These objects may

be stateful. Any object created by an actor is said to be *owned* by that actor, and forever remains contained in that actor. Objects owned by one actor may hold references to individual objects owned by other actors (i.e. objects contained by an actor may be referenced from outside of the actor, they are not necessarily private).

- There is no blocking synchronization primitive: both the sending and receiving of messages between actors happens asynchronously. Contrary to e.g. Erlang or Scala actors, there is no direct equivalent to the `receive` statement that suspends an actor until a matching message arrives. Instead, message reception happens implicitly by invoking a method on an object.

Thus, actors are not represented as individual objects, but rather as *a collection of* objects that all share a single event loop which executes their code. That event loop has a single message queue, containing messages to be delivered to one of its owned objects. The event loop perpetually takes the first message from the message queue and invokes the corresponding method of the object denoted as the receiver of the message. This method is then run to completion, without interleaving any other events. Consider the following example:

```
def makeAccount(balance) {
  object: {
    def withdraw(amnt) { balance := balance - amnt };
    def deposit(amnt) { balance := balance + amnt };
  }
}
def b1 := makeAccount(50);
def b2 := makeAccount(20);
```

By default, there is a single "main" AmbientTalk actor that executes all top-level code. In this example, the main actor creates (and thus owns) two account objects `b1` and `b2`. Any external requests to `withdraw` or `deposit` from these accounts will be executed without interleaving.

The process of dequeuing a message (such as `withdraw` or `deposit`) from the actor's queue and executing the corresponding method to completion is called a *turn*. In between turns, the runtime stack of an actor is always empty. Turns are the basic unit of "event interleaving" in AmbientTalk: while executing a turn, no other events can affect the actor's heap. In event-loop frameworks, this is sometimes called *run-to-completion* semantics, since every event is fully processed before processing the next. This avoids data races on the mutable state of objects owned by an actor.

Only an object's owning actor may directly execute its methods. Objects owned by the same actor may communicate using ordinary, sequential method invocation or using asynchronous message passing. AmbientTalk borrows from the E language the syntactic distinction between sequential method invocation (expressed as `o.m()`) and asynchronous message sending (expressed as `o<-m()`).

For example, since the main actor owns both `b1` and `b2`, it may atomically transfer funds from one account to the other by executing:

```
b1.withdraw(10);
b2.deposit(10);
```

It may also decide to send these messages asynchronously:

```
b1<-withdraw(10);
b2<-deposit(10);
```

This enqueues the requests to `withdraw` and `deposit` in the main actor's own message queue. However, the programmer should now be aware that other messages may happen to arrive after `withdraw` but before `deposit` was scheduled. In other words, the transfer is no longer atomic, which may or may not be a problem, depending on application requirements.

5.2 Far References

It is possible for objects owned by one actor to hold references to individual objects owned by other actors. Such references that span different actors are named *far references* (the terminology stems from E [MTS05]) and only allow asynchronous access to the referenced object. This ensures by design that all communication between actors is asynchronous. Trying to perform a sequential method invocation on a far reference provokes a runtime exception.

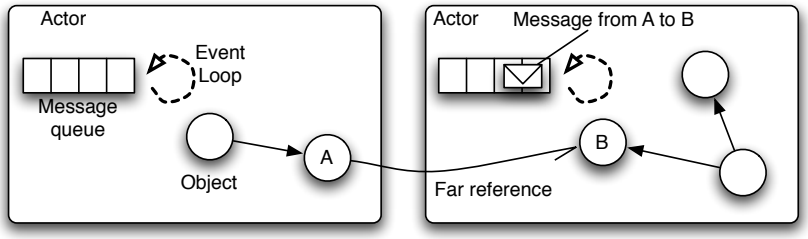

Figure 1: AmbientTalk actors as communicating event loops.

Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the event loop activity of each actor which perpetually takes the next message from its message queue and executes the corresponding method on one of its owned objects.

To illustrate far references, consider the following example. Our main actor spawns a new actor and decides to share its account objects with this new actor:

```
def helper := actor: {
  def transfer(from,to) {
    from<-withdraw(10);
    to<-deposit(10);
  }
};
helper<-transfer(b1, b2);
```

The expression `actor: {...}` spawns a new actor. The new actor immediately creates a new object, as if by evaluating `object: {...}`. This object, call it `o`, acts as the actor's public interface. The `actor:` expression immediately evaluates to a far reference to `o`, which is stored in the `helper` variable.

The main actor then sends the `transfer` message via a far reference to `o`. Messages sent via a far reference to an object are enqueued in the message queue of the object's owner for later processing. Hence, the `transfer` message will later be dequeued and executed by the new actor, not by the main actor.

5.3 Asynchronous Message Passing and Isolates

When sending an asynchronous message to an object that is owned by the same actor, the message's parameters are passed *by reference*, exactly as is the case with regular sequential method invocations. When sending a message across a far reference to another actor, objects are instead parameter-passed *by far reference*: the parameters of the invoked method are bound to far references to the original objects.

Take another look at the previous example. The main actor passes `b1` and `b2` as arguments to an asynchronous `transfer` message. When the helper actor executes the `transfer` method, `from` and `to` will be bound to far references to `b1` and `b2` respectively. Note that the helper actor must use asynchronous message passing (via `<-`) to perform the `withdraw` and `deposit` operations. Since it does not own the account objects, it cannot directly invoke their methods.

There is one exception to the above parameter-passing rules: objects declared as *isolates* (via the expression `isolate:{...}` as opposed to `object:{...}`) are passed by (deep) copy rather than by far reference. Objects can only be declared as isolates if all of their methods are closed (i.e., do not contain references to lexically free variables). This ensures that such objects are isolated from their scope of definition (hence their name), allowing their methods to be safely executed in other actors. This restriction also ensures that isolates can be serialized without having to transitively serialize the value of any lexically captured variables. The benefit of isolates is that the recipient actor will receive its own local copy of the isolate, avoiding further remote communication.

5.4 Non-blocking Futures

By default, asynchronous message sends do not return a meaningful value (to be more precise, they return `null`). Often, an object that makes an asynchronous request is interested in a later reply. For instance, in our previous example, what if the main actor wants to know when the `transfer` performed by the helper actor was completed? This can be accomplished as follows:

```
def future := helper<-transfer(b1, b2)@TwoWay;
```

Any AmbientTalk message may be annotated with the `TwoWay` type tag to indicate that the message should return a *future*. A future is a placeholder for the later return value, which may not yet be available. Initially, the future is said to be *unresolved*.

The future gives us a handle on the return value, but is not itself the return value. One can register a callback with a future, which is executed when the future becomes resolved, and is passed the actual return value of the message:

```
when: future becomes: { |ack|
  // execution is postponed until future is resolved
  system.println("Transfer performed");
} catch: { |exception|
  system.println("Transfer failed");
};
// code hereafter is always executed first, even if future is already resolved
```

The `when:becomes:catch:` function takes a future and two blocks (a callback and an errback) as arguments, and registers these blocks with the future, as if they were listener objects. If the asynchronously invoked method returns a value, the future is *resolved*, and the callback is called with the return value (in the above example, the `transfer` method just returns `null`, so `ack` will be `null` as well, serving only as

an acknowledgement). If the method instead raises an exception, the corresponding future becomes *ruined* and the errback is called with the exception. The errback is analogous to a `catch`-clause in a regular sequential `try-catch` statement.

Even if `when:becomes:catch:` is called on a future that is already resolved, the callback or errback is never immediately invoked, but instead always scheduled for eventual execution in the message queue of the actor that created the block. This ensures that the callback or errback is always executed in its own separate turn, and that the execution is properly serialized w.r.t. other messages processed by the actor.

Returning to our example, we still have not quite successfully synchronized on the actual transfer of the money: when `future` resolves, all we actually know is that the `transfer` method was executed. But since the transfer method itself performs asynchronous requests, completion of the transfer method does not imply completion of the `withdraw` and `deposit` messages. This type of transitive asynchronous dependencies comes up sufficiently often that AmbientTalk futures provide support for it. It is possible to resolve a future f_1 with *another* future f_2, establishing a dataflow dependency among them: if f_2 later becomes resolved with a non-future value v, then eventually f_1 will also become resolved with v. Returning to our example, we need to change the `transfer` method as follows:

```
def helper := actor: {
  def transfer(from,to) {
    from<-withdraw(10);
    def f2 := to<-deposit(10)@TwoWay; // note the new annotation
    f2
  }
};
def future := helper<-transfer(b1, b2)@TwoWay;
when: future becomes: { |ack|
  system.println("Transfer performed");
} catch: { |e| ... }
```

The `transfer` method now returns a future `f2`, rather than `null`. The outer `future` will be resolved with this future `f2`. The callback will be triggered only when the `deposit` message, sent while executing the `transfer` method, has itself returned.

While this particular example is correct, the code for `transfer` in general is not: our synchronization only works because we know `from` and `to` refer to account objects owned by the same actor. Since AmbientTalk actors enqueue messages in FIFO order, we know that if the `deposit` method was executed on `to`, the `withdraw` method was also executed on `from`, since it was enqueued earlier in the same actor. In the general case where `from` and `to` may refer to objects in different actors, we can no longer make that assumption.

AmbientTalk has a number of auxiliary functions that operate on futures. One such function is `group:`. This function expects an array of futures `[f1,f2,...]` and returns a new "composite" future `f`. `f` is resolved with the array `[v1,v2,...]` when and only when `f1,f2,...` have all resolved to values `v1,v2,...`. If any of the argument futures is ruined with an exception, `f` becomes ruined with that same exception. If one thinks of futures as booleans with states resolved and ruined, then `group:` is the equivalent of the logical AND operator. Armed with `group:` we can apply the proper synchronization:

```
def helper := actor: {
  def transfer(from,to) {
    def f1 := from<-withdraw(10)@TwoWay;
    def f2 := to<-deposit(10)@TwoWay;
    group: [f1,f2]
```

```
  }
};
def future := helper<-transfer(b1, b2)@TwoWay;
when: future becomes: { |ack|
  system.println("Transfer performed");
}
```

The callback is now triggered only after both `f1` and `f2` have resolved.

Futures as Far References AmbientTalk futures are also far references to their eventual value: one can send asynchronous messages to the future, and these are automatically forwarded to their value. As long as the future is unresolved, the messages are accumulated at the future. When the future is resolved, these accumulated messages are forwarded to the resolved value. If the future is ruined, any futures associated with accumulated messages are ruined with the same exception. This is the asynchronous equivalent of an exception propagating up the call stack.

Conditional Synchronization So far, the only way to obtain a future has been to send an asynchronous message annotated as `@TwoWay`. In addition, these futures are automatically resolved with the return value of the corresponding method. Sometimes, this rigid pattern of using futures is insufficient: it may be that the resolved value of a future depends on run-time conditions known only at a later stage in the program. For instance, consider a bounded buffer with `get()` and `put(v)` methods. When the buffer is empty, it may want its `get()` method to return an unresolved future, to be resolved later when a producer sends a `put(v)` message.

To facilitate such "conditional synchronization" [BGL98] patterns, it is possible to explicitly create and resolve futures:

```
def [future, resolver] := makeFuture();
```

The call to `makeFuture` returns two values: a fresh, unresolved `future` object, and a paired `resolver` object. The creator can pass the `future` object around freely to third parties, while retaining a reference to the `resolver` object. At a later time, when the value of the future is known, the creator can invoke `resolver.resolve(value)` to resolve the paired future. This will also trigger any pending callbacks registered with the future.

5.5 Concurrency Properties

The key concurrency properties provided by communicating event loops are:

No data races Since every object is owned by exactly one actor, and since actors process incoming messages for their owned objects sequentially, data races on the state of objects are avoided: there is at most one concurrent activity that can read or write to their fields. While low-level data races are prevented, race conditions at the level of messages are still possible (e.g. unexpected message interleavings).

No deadlocks Since all communication between actors is purely asynchronous, deadlocks in the traditional sense are avoided. In particular, message reception is implicit and fully asynchronous (an actor never suspends during a turn). Also, AmbientTalk futures are fully non-blocking. Contrary to most future abstractions, they do not support a blocking `get()` method to await the future's value

synchronously. One must use the `when:becomes:catch:` function to await the future's value in a non-blocking manner. Note that lost progress bugs are still possible, e.g. a future with pending callbacks may never be resolved, so its callbacks will never fire.

6 Distributed AmbientTalk

We now turn to AmbientTalk's features specifically geared towards distributed programming in mobile ad hoc networks.

A key aspect of distributed programming in AmbientTalk is that (objects in) actors running on different devices can communicate with each other, as if those actors were running on the same device. There are of course differences in terms of failure handling, but the programming model remains identical. When an object a acquires a far reference to an object b in another actor, we call b a *remote object* (from a's point of view), regardless of whether both actors are running on the same device.

6.1 Service Discovery

We have previously shown that objects can acquire a new far reference to a remote object by simply passing an object as a parameter into or as a return value from a message sent via an existing far reference. However, this requires some *initial* far reference to an object in the remote actor. How is this process bootstrapped?

AmbientTalk uses a publish/subscribe service discovery protocol. A publication corresponds to an object advertising itself by means of a type tag. The type tag serves as a *topic* known to both publishers and subscribers [EFGK03]. A subscription is made by registering a callback block on a type tag. The callback will be triggered whenever an object advertised with that tag is detected in the network.

An object that advertises itself is said to be *exported*. Once exported, an object becomes a globally accessible entry-point. In most distributed systems, exported objects are identified by means of a URL and a UUID, or similar such global identifiers. However, URLs rely on infrastructure (name servers), which cannot always be relied upon in a mobile ad hoc network. In addition, in mobile P2P applications, one application is often interested in *any* other application with which it can partner, not necessarily a specific application. Thus, mobile P2P applications are more interested in a *type* of service than a particular unique instance of a service.

We use type tags to provide a description of what kinds of services an object provides to remote objects. We make the explicit assumption that all devices in the network attribute the same meaning to each type tag, i.e. we assume they use a common classification scheme.

Assume a mobile P2P application named `MatchMaker` that wants to pair up with other applications of the same type. This application exchanges user profiles and alerts the user when a matching profile is found. The `MatchMaker` application exports an object serving as its publicly accessible entry-point, as shown below.

```
deftype MatchMaker;
def myEndPoint := object: {
  def exchange(profile) { ... }
  def alertMatch(profile) { ... }
};
def pub := export: myEndPoint as: MatchMaker;
```

Once the `myEndPoint` object is exported, it can be discovered by other actors. The `export:as:` function returns an object `pub` that can be used to take the exported object offline again, by invoking `pub.cancel()`.

To discover remote endpoints of peer applications, a `MatchMaker` application can subscribe a callback to be notified whenever a matching endpoint is discovered in the network:

```
whenever: MatchMaker discovered: { |remoteEndPoint|
  remoteEndPoint<-exchange(myProfile);
  ...
};
```

The `whenever:discovered:` function takes as arguments a type tag and a block that serves as a callback. Every time an object with a matching type tag is discovered by the language runtime, an invocation of this callback is enqueued in the actor owning the block. The `remoteEndPoint` argument to the block is bound to a far reference pointing to the `myEndPoint` object of a peer `MatchMaker` application.

Similar to the `export:as:` function, the `whenever:discovered:` function returns a subscription object whose `cancel()` method can be used to cancel the registration of the callback.

6.2 Far References and Partial Failures

Because objects residing on different devices are necessarily owned by different actors, far references are the only kind of object reference that can span across different devices. By design, this ensures that all distributed communication is asynchronous. This strict adherence to asynchronous distributed communication has two advantages in wireless networks:

- First, latency in wireless networks is still more significant than in wired networks. Asynchronous communication helps to hide latency, enabling applications to perform useful work, or remain responsive, even while sending and receiving messages.
- Second, as noted previously, connections among roaming mobile devices are often volatile. Asynchronous communication facilitates communication along such intermittent connections via buffering. When sender and receiver are disconnected, outgoing messages can be buffered and retransmitted when the connection is restored. This is like sending e-mail while working offline.

AmbientTalk's far references make use of such buffering to be resilient to network disconnections by default. Returning to our previous example, when a `MatchMaker` application discovers a peer, it obtains a far reference `remoteEndPoint` to communicate further. Should the peer application disconnect at that point, the `exchange` message will be buffered within the reference. When the network partition is eventually restored, the far reference automatically retransmits the `exchange` message. Hence, messages sent to far references are never lost, regardless of the internal connection state of the reference.

Of course, not all network partitions are transient. Some will be permanent, or sufficiently long-lasting to require application-level failure handling. To this end, AmbientTalk makes use of leasing [GC89]. An object can be configured such that any far reference that points to it provides access for only a limited period of time (the lease

period). Instead of exporting the `myEndPoint` object directly, the `MatchMaker` application can export a *lease* for it:

```
def myEndPoint := object: {
  // as before
};
def leasedEndPoint := lease: myEndPoint for: 2.minutes;
def pub := export: leasedEndPoint as: MatchMaker;
```

The function `lease:for:` expects an object and a duration (here, 2 minutes) and returns a leased proxy for the object. Any far reference created to the leased proxy remains valid for at most 2 minutes from the time the reference was first created. At the discretion of the creator of the lease, the lease can be renewed, prolonging access to the object. By default, the lease is renewed every time a message arrives at the proxy.

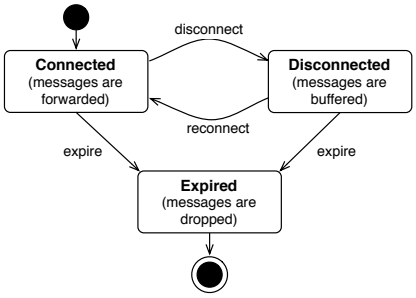

Figure 2: State diagram of a (leased) far reference.

Figure 2 summarizes the different states a far reference can be in. When the far reference is connected and the lease has not yet expired, it forwards messages to the remote object. While disconnected, messages are accumulated in the reference, as previously explained. When the lease expires, access to the remote object is permanently revoked and the far reference itself becomes expired. Any message sent to an expired reference is discarded (not buffered), and any future associated with this message is ruined with an appropriate exception. Far references to non-leased objects are like leased references whose lease period is infinite.

Both endpoints of a far reference can register callbacks to be invoked upon expiration, e.g. to schedule clean-up actions. Leased references facilitate automatic distributed memory management: once all far references to an object have expired, the object can be taken offline, becoming subject to garbage collection if it is no longer locally referenced. Without leasing, a single disconnected far reference could keep an object online forever.

7 Operational Semantics

We present a small step operational semantics of a subset of AmbientTalk, named AT-LITE. Our goal here is to provide a precise definition of AmbientTalk's concurrency model. The presented semantics is based primarily on that of JCobox [SPH10], but adapted for a dynamically typed, classless language, and modified to precisely reflect AmbientTalk's communicating event loops model with non-blocking futures.

The AT-LITE subset contains actors, objects, isolates (pass-by-copy objects), blocks (functions), non-blocking futures and asynchronous message sending. In Section 7.4

we extend AT-LITE with service discovery, enabling objects in different actors to discover one another. In Section 7.5, we introduce the notion of disconnected actors and fault-tolerant messaging between actors. AT-LITE does not model AmbientTalk's object inheritance using prototype-based delegation, its support for trait-based composition (which is formalised elsewhere [VCBDM09]), reflection, exceptions and JVM interoperability.

The semantics of AT-LITE is implemented in PLT Redex [FFF09], allowing AT-LITE programs to be executed. Our PLT Redex implementation is available from `http://soft.vub.ac.be/~cfscholl/AT-Redex-Model.zip`.

7.1 Syntax

$$
\begin{array}{rcl}
e \in E \subseteq \textbf{Expr} & ::= & \text{self} \mid x \mid \text{null} \mid e\,;\,e \mid \lambda x.e \mid e(\overline{e}) \mid \text{let}\, x = e \,\text{in}\, e \mid e.f \mid e.f := e \\
& \mid & e.m(\overline{e}) \mid \text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \mid \text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
& \mid & \text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \mid \text{let}\, x_f, x_r = \text{future}\,\text{in}\, e \mid \text{resolve}\, e\, e \\
& \mid & e \leftarrow m(\overline{e}) \mid e \leftarrow_f m(\overline{e}) \mid \text{when}(e \rightarrow x)\{e\}
\end{array}
$$

$$x, x_f, x_r \in \textbf{VarName}, f \in \textbf{FieldName}, m \in \textbf{MethodName}$$

Figure 3: Abstract Syntax of AT-LITE.

AT-LITE features both functional and imperative object-oriented elements. The functional elements descend directly from the λ-calculus. Anonymous functions are denoted by $\lambda x.e$ and correspond to AmbientTalk blocks. Variable lookup in AT-LITE is lexically scoped. Local variables can be introduced via $\text{let}\, x = e \,\text{in}\, e$.

The imperative object-oriented elements stem from object-based (i.e. classless) calculi [AC96]. AT-LITE features `object {...}` and `isolate {...}` literal expressions to define fresh, anonymous objects. These literals consist of a sequence of field and method declarations. Fields may be accessed and updated. Methods may be invoked either synchronously via $e.m(\overline{e})$ or asynchronously via $e \leftarrow m(\overline{e})$.

In the scope of a method body, the pseudovariable `self` refers to the enclosing object literal. `self` cannot be used as a parameter name in methods or redefined using `let`.

New actors can be spawned using the `actor {...}` literal expression. This creates a new object with the given fields and methods in a fresh actor that executes in parallel. Actor and isolate literals may not refer to lexically enclosing variables, apart from the `self`-pseudovariable. That is, for all field initialiser and method body expressions e in such literals, the set of free variables $FV(e) \subseteq \{\text{self}\}$. Isolates and actors are thus literally "isolated" from their surrounding lexical scope, allowing their subexpressions to be evaluated independent of the lexical scope in which they were defined.

New futures can be created explicitly using the expression $\text{let}\, x_f, x_r = \text{future}\,\text{in}\, e$. This binds a fresh future to the variable x_f and a fresh, paired resolver object to x_r. A resolver object denotes the right to assign a value to its paired future. The expression $\text{resolve}\, x_r\, e$ resolves the future x_f via its paired resolver x_r with the value of e. The value of a future x_f can be awaited using the expression $\text{when}(x_f \rightarrow x)\{e\}$. When

the future becomes resolved with a value v, the expression e is evaluated with x bound to v.

AT-LITE supports two forms of asynchronous message passing. Expressions of the form $e \leftarrow m(\overline{e})$ denote one-way asynchronous message sends that do not return a useful value. If a return value is expected, the expression $e \leftarrow_f m(\overline{e})$ denotes a two-way asynchronous message send that immediately returns a future for the result of invoking the method m. This is the equivalent of messages annotated with the `@TwoWay` tag in AmbientTalk.

7.1.1 Syntactic Sugar

A number of AT-LITE expressions can be defined in terms of a desugaring (local transformation), as shown in Figure 4.

$$
\begin{array}{rcll}
e\ ;\ e' & \stackrel{\text{def}}{=} & \text{let } x = e \text{ in } e' & x \notin \text{FV}(e') \\
\lambda x.e & \stackrel{\text{def}}{=} & \text{let } x_{\text{self}} = \text{self in object } \{ & x_{\text{self}} \notin \text{FV}(e) \\
 & & \ \text{apply}(x)\{[x_{\text{self}}/\text{self}]e\} & \\
 & & \} & \\
e(\overline{e}) & \stackrel{\text{def}}{=} & e.\text{apply}(\overline{e}) & \\
e \leftarrow_f m(\overline{e}) & \stackrel{\text{def}}{=} & \text{let } x_f, x_r = \text{future in} & x_f, x_r \notin \text{FV}(e) \cup \text{FV}(\overline{e}) \\
 & & \ e \leftarrow m_f(\overline{e} \cdot x_r)\ ;\ x_f & \\
\text{when}(e \rightarrow x)\{e'\} & \stackrel{\text{def}}{=} & \text{let } x_f, x_r = \text{future in} & x_f, x_r \notin \text{FV}(e) \cup \text{FV}(e') \\
 & & \ \text{let } x_c = \lambda x.(x_r.\text{resolve}_\mu(e')) \text{ in} & x_c \notin \text{FV}(e) \\
 & & \ \ e \leftarrow \text{register}_\mu(x_c)\ ;\ x_f & \\
\text{resolve } e\ e' & \stackrel{\text{def}}{=} & \text{let } x_r = e \text{ in} & x_r \notin \text{FV}(e') \\
 & & \ \text{let } x_c = \lambda x.(x_r \leftarrow \text{resolve}_\mu(x)) \text{ in} & x_c \notin \text{FV}(e') \\
 & & \ \ e' \leftarrow \text{register}_\mu(x_c) &
\end{array}
$$

Figure 4: AT-LITE syntactic sugar.

It is well-known that functions can be expressed in terms of objects and vice-versa. AT-LITE functions (like AmbientTalk blocks) are defined as objects with a single method called `apply`. The substitution $[x_{\text{self}}/\text{self}]e$ is necessary to ensure that within function bodies nested inside object methods, the `self`-pseudovariable remains bound to the object enclosing the function, and not to the object representing the function. Function application $e(\overline{e})$ is desugared into invoking an object's `apply` method.

A two-way message send $e \leftarrow_f m(\overline{e})$ is syntactic sugar for a simple one-way message send that carries a fresh resolver object x_r, added as a hidden last argument. The message m is marked m_f, serving as a signal for the recipient actor that it needs to pass the result of the method invocation to x_r. The value of a two-way message send expression is the future x_f corresponding to the passed resolver x_r.

The expression $\text{when}(e \rightarrow x)\{e'\}$ is used to await the value of a future. It is syntactic sugar for registering a callback function x_c with the future. The expression as a whole returns a *dependent* future x_f that will become resolved with the expression e' when the future denoted by e eventually resolves.

The expression $\text{resolve } e\ e'$ is used to resolve a future with a value, where e must

reduce to a resolver and e' to any value. If e' reduces to a non-future value, the callback function x_c will be called with x bound to the value of e'. If e' reduces to a future value, the callback function will be called later, with x bound to the resolved value of the future. Thus, this definition ensures that futures can only be truly resolved with non-future values.

The desugaring of "when" and "resolve" make use of special messages named resolve_μ and register_μ. The μ (for "meta") suffix identifies these messages as special meta-level messages that should be interpreted differently by actors. A regular AT-LITE program cannot fabricate these messages other than via the "when" and "resolve" expressions.

7.2 Semantic Entities

$$
\begin{array}{rcll}
K \in \textbf{Configuration} & ::= & A & \text{Configurations} \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, O, Q, e \rangle & \text{Actors} \\
\textbf{Object} & ::= & \mathcal{O}\langle \iota_o, t, F, M \rangle & \text{Objects} \\
t \in \textbf{Tag} & ::= & \text{O} \mid \text{I} & \text{Object tags} \\
\textbf{Future} & ::= & \mathcal{F}\langle \iota_f, Q, v \rangle & \text{Futures} \\
\textbf{Resolver} & ::= & \mathcal{R}\langle \iota_r, \iota_f \rangle & \text{Resolvers} \\
\text{m} \in \textbf{Message} & ::= & \mathcal{M}\langle v, m, \overline{v} \rangle & \text{Messages} \\
Q \in \textbf{Queue} & ::= & \overline{\text{m}} & \text{Queues} \\
M \subseteq \textbf{Method} & ::= & m(\overline{x})\{e\} & \text{Methods} \\
F \subseteq \textbf{Field} & ::= & f := v & \text{Fields} \\
v \in \textbf{Value} & ::= & r \mid \text{null} \mid \epsilon & \text{Values} \\
r \in \textbf{Reference} & ::= & \iota_a.\iota_o \mid \iota_a.\iota_f \mid \iota_a.\iota_r & \text{References} \\
e \in E \subseteq \textbf{Expr} & ::= & \ldots \mid r & \text{Runtime Expressions}
\end{array}
$$

$$
o \in O \subseteq \textbf{Object} \cup \textbf{Future} \cup \textbf{Resolver}
$$
$$
\iota_a \in \mathbf{ActorId}, \iota_o \in \mathbf{ObjectId}
$$
$$
\iota_f \in \mathbf{FutureId} \subset \mathbf{ObjectId}, \iota_r \in \mathbf{ResolverId} \subset \mathbf{ObjectId}
$$

Figure 5: Semantic entities of AT-LITE.

AT-LITE semantic entities are shown in Figure 5. Caligraphic letters like $\mathcal{F}$ and $\mathcal{M}$ are used as "constructors" to distinguish the different semantic entities syntactically. Regular uppercase letters like F and M denote sets or sequences. Actors, futures, resolvers and objects each have a distinct address or identity, denoted ι_a, ι_f, ι_r and ι_o respectively.

The state of an AT-LITE program is represented by a configuration K, which is a set of concurrently executing actors. Each actor is an event loop consisting of an identity ι_a, a heap O denoting the set of objects, futures and resolvers *owned* by the actor, a queue Q containing a sequence of messages to be processed, and the expression e that the actor is currently executing.

Objects consist of an identity ι_o, a tag t and a set of fields F and methods M. The tag t is used to distinguish objects from isolates, with $t = \text{O}$ denoting an object and $t = \text{I}$ denoting an isolate. Isolates are parameter-passed by-copy rather than by-reference in remote message sends, but otherwise behave the same as regular objects.

An AT-LITE future is a first-class placeholder for an asynchronously awaited value. Futures consist of an identity ι_f, a queue of pending messages Q and a resolved value v. A future is initially *unresolved*, in which case its resolved value v is set to a unique empty value ϵ. While the future is unresolved, any messages sent to the future are queued up in Q. When the future becomes *resolved*, all messages in Q are forwarded to the resolved value v and the queue is emptied. We do not model AmbientTalk's support for ruined futures and asynchronous propagation of exceptions.

A resolver object denotes the right to assign a value to its unique paired future. Resolvers consist of an identity ι_r and the identity of their paired future ι_f. The resolver is the only means through which a future can be resolved with a value.

Messages are triplets consisting of a receiver value v, a method name m and a sequence of argument values $\overline{v}$. They denote asynchronous messages that are enqueued in the message queue of actors or futures.

All object references consist of a global component ι_a that identifies the actor owning the referenced value, and a local component ι_o, ι_f or ι_r. The local component indicates that the reference refers to either an object, a resolver or a future. We define $\mathbf{FutureId}$ and $\mathbf{ResolverId}$ to be a subset of $\mathbf{ObjectId}$ such that a reference to a future or a resolver is also a valid object reference. As such, $\iota_a.\iota_o$ can refer to either an object, a resolver or a future, but $\iota_a.\iota_f$ can refer only to a future.

Our reduction rules operate on "runtime expressions", which are simply all expressions e including references r, as a subexpression may reduce to a reference before being reduced further.

7.3 Reduction Rules

7.3.1 Evaluation Contexts

We use evaluation contexts [FH92] to indicate what subexpressions of an expression should be fully reduced before the compound expression itself can be further reduced:

$$\begin{array}{rcl} e_\Box & ::= & \Box \mid \text{let } x \ = \ e_\Box \text{ in } e \mid e_\Box.f \mid e_\Box.f := e \mid v.f := e_\Box \mid e_\Box.m(\overline{e}) \mid v.m(\overline{v}, e_\Box, \overline{e}) \\ & \mid & e_\Box \leftarrow m(\overline{e}) \mid v \leftarrow m(\overline{v}, e_\Box, \overline{e}) \end{array}$$

$e_\Box$ denotes an expression with a "hole". Each appearance of $e_\Box$ indicates a subexpression with a possible hole. The intent is for the hole to identify the next subexpression to reduce in a compound expression. The notation $e_\Box[e]$ indicates that the expression e is part of a compound expression $e_\Box$, and should be reduced first before the compound expression can be reduced further.

7.3.2 Notation

Actor heaps O are sets of objects, resolvers and futures. To lookup and extract values from a set O, we use the notation $O = O' \cup \{o\}$. This splits the set O into a singleton set containing the desired object o and the disjoint set $O' = O \setminus \{o\}$. The notation $Q = Q' \cdot \mathrm{m}$ deconstructs a sequence Q into a subsequence Q' and the last element m. We represent queues as sequences of messages that are processed right-to-left, meaning that the last message in the sequence is the first to be processed. We denote both the empty set and the empty sequence as $\emptyset$.

7.3.3 Evaluation Rules

Our semantics is defined in terms of a relation on configurations, $K \rightarrow K'$. It is implicitly parametrized by a fixed underlying AT-LITE program. The rules defining the relation are split into two parts: actor-local rules $a \rightarrow_a a'$ and global rules $K \rightarrow_k K'$. This makes it explicit which steps can be executed in isolation within a single actor a, and which require interaction between different actors in a configuration K.

Both actor-local and global rules can be applied non-deterministically, which gives rise to concurrency. We do not yet consider actors distributed across different devices, connected by a network, until Section 7.5. For now, we consider all actors to remain permanently connected with each other.

Actor-local reductions Actors operate by perpetually taking the next message from their message queue, transforming the message into an appropriate expression to evaluate, and then evaluate (reduce) this expression to a value. When the expression is fully reduced, the next message is processed. As discussed previously, the process of reducing such a single expression to a value is called a *turn*. It is not possible to suspend a turn and start processing another message in the middle of a reduction.

The only valid state in which an actor cannot be further reduced is when its message queue is empty, and its current expression is fully reduced to a value. The actor then sits idle until it receives a new message. If an actor is reducing a compound expression, and finds no applicable actor-local reduction rule to reduce it further, the actor is stuck. This signifies an error in the program.

$$
\begin{array}{rcl@{\qquad}rcl}
[v/x]x' & = & x' & [v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{e\} \text{ if } x \in \overline{x} \\
[v/x]x & = & v & [v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{[v/x]e\} \text{ if } x \notin \overline{x} \\
[v/x]e.f & = & ([v/x]e).f & [v/x]e.f := e & = & ([v/x]e).f := [v/x]e \\
[v/x]\text{null} & = & \text{null} & [v/x]e.m(\overline{e}) & = & [v/x]e.m([v/x]\overline{e}) \\
[v/x]r & = & r & [v/x]e \leftarrow m(\overline{e}) & = & [v/x]e \leftarrow m([v/x]\overline{e})
\end{array}
$$

$$
\begin{array}{rcl}
[v/x]\text{let } x' = e \text{ in } e & = & \text{let } x' = [v/x]e \text{ in } [v/x]e \\
[v/x]\text{let } x = e \text{ in } e & = & \text{let } x = [v/x]e \text{ in } e \\
[v/x]\text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
[v/x]\text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
[v/x]\text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f := [v/x]e}, \overline{[v/x]m(\overline{x})\{e\}}\} \text{ if } x \neq \text{self} \\
[v/\text{self}]\text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
[v/x]\text{let } x_f, x_r = \text{future in } e & = & \text{let } x_f, x_r = \text{future in } [v/x]e \\
[v/x]\text{let } x, x_r = \text{future in } e & = & \text{let } x, x_r = \text{future in } e \\
[v/x]\text{let } x_f, x = \text{future in } e & = & \text{let } x_f, x = \text{future in } e
\end{array}
$$

Figure 6: Substitution rules: x denotes a variable name or the pseudovariable self, v denotes a value.

We now summarize the actor-local reduction rules in Figure 7:

- LET: a "let"-expression simply substitutes the value of x for v in e according to the substitution rules outlined in Figure 6.

(LET)

$$\mathcal{A}\langle \iota_a, O, Q, e_\Box[\text{let } x = v \text{ in } e]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e_\Box[[v/x]e]\rangle$$

(NEW-OBJECT)

$$\frac{\iota_o \text{ fresh} \qquad o = \mathcal{O}\langle \iota_o, \mathrm{o}, \overline{f := \text{null}}, \overline{m(\overline{x})\{e'\}}\rangle \quad r = \iota_a.\iota_o}{\mathcal{A}\langle \iota_a, O, Q, e_\Box[\text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e'\}}\}]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O \cup \{o\}, Q, e_\Box[\overline{r.f := [r/\text{self}]e}; r]\rangle}$$

(NEW-ISOLATE)

$$\frac{\iota_o \text{ fresh} \qquad o = \mathcal{O}\langle \iota_o, \mathrm{I}, \overline{f := \text{null}}, \overline{m(\overline{x})\{e'\}}\rangle \quad r = \iota_a.\iota_o}{\mathcal{A}\langle \iota_a, O, Q, e_\Box[\text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e'\}}\}]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O \cup \{o\}, Q, e_\Box[\overline{r.f := [r/\text{self}]e}; r]\rangle}$$

(INVOKE)

$$\frac{\mathcal{O}\langle \iota_o, t, F, M\rangle \in O \qquad r = \iota_a.\iota_o \qquad m(\overline{x})\{e\} \in M}{\mathcal{A}\langle \iota_a, O, Q, e_\Box[r.m(\overline{v})]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e_\Box[[r/\text{self}][\overline{v}/\overline{x}]e]\rangle}$$

(FIELD-ACCESS)

$$\frac{\mathcal{O}\langle \iota_o, t, F, M\rangle \in O \qquad f := v \in F}{\mathcal{A}\langle \iota_a, O, Q, e_\Box[\iota_a.\iota_o.f]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e_\Box[v]\rangle}$$

(FIELD-UPDATE)

$$\frac{O = O' \uplus \{\mathcal{O}\langle \iota_o, t, F \uplus \{f := v'\}, M\rangle\} \qquad O'' = O' \cup \{\mathcal{O}\langle \iota_o, t, F \cup \{f := v\}, M\rangle\}}{\mathcal{A}\langle \iota_a, O, Q, e_\Box[\iota_a.\iota_o.f := v]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O'', Q, e_\Box[v]\rangle}$$

(MAKE-FUTURE)

$$\frac{\iota_f, \iota_r \text{ fresh} \qquad O' = O \cup \{\mathcal{F}\langle \iota_f, \emptyset, \epsilon\rangle, \mathcal{R}\langle \iota_r, \iota_f\rangle\}}{\mathcal{A}\langle \iota_a, O, Q, e_\Box[\text{let } x_f, x_r = \text{future in } e]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O', Q, e_\Box[[\iota_a.\iota_f/x_f][\iota_a.\iota_r/x_r]e]\rangle}$$

(LOCAL-ASYNCHRONOUS-SEND)

$$\mathcal{A}\langle \iota_a, O, Q, e_\Box[\iota_a.\iota_o \leftarrow m(\overline{v})]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O, \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle \cdot Q, e_\Box[\text{null}]\rangle$$

(PROCESS-MESSAGE)

$$\frac{\iota_o \notin \textbf{FutureId} \qquad e = \mathit{process}(\iota_a.\iota_o, m, \overline{v})}{\mathcal{A}\langle \iota_a, O, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle, v\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e\rangle}$$

(PROCESS-MSG-TO-FUTURE)

$$\frac{O = O' \uplus \{\mathcal{F}\langle \iota_f, Q', v'\rangle\} \qquad (\mathrm{m}, e) = \mathit{store}(m, \overline{v}, v')}{\mathcal{A}\langle \iota_a, O, Q \cdot \mathcal{M}\langle \iota_a.\iota_f, m, \overline{v}\rangle, v\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O' \cup \{\mathcal{F}\langle \iota_f, \mathrm{m} \cdot Q', v'\rangle\}, Q, e\rangle}$$

(RESOLVE)

$$\frac{\mathcal{R}\langle \iota_r, \iota_f\rangle \in O \qquad O = O' \uplus \{\mathcal{F}\langle \iota_f, Q', \epsilon\rangle\} \qquad v \neq \iota_{a'}.\iota_{f'}}{\mathcal{A}\langle \iota_a, O, Q, e_\Box[\iota_a.\iota_r.\text{resolve}_\mu(v)]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O' \cup \{\mathcal{F}\langle \iota_f, \emptyset, v\rangle\}, Q, e_\Box[\mathit{fwd}(v, Q')]\rangle}$$

Figure 7: Actor-local reduction rules.

- NEW-OBJECT, NEW-ISOLATE: these rules are identical except for the tag of the fresh object, which is set to O for objects and I for isolates. Evaluating an object or literal expression adds a new object to the actor's heap. The new object's fields are initialised to `null`. The literal expression reduces to a sequence of field update expressions. The `self` pseudovariable within these field update expressions refers to the new object. The last expression in the reduced sequence is a reference r to the new object.

- INVOKE: a method invocation looks up the method m in the receiver object $\iota_a.\iota_o$ and reduces the method body expression e with appropriate values for the parameters $\overline{x}$ and the pseudovariable `self`. It is *only* possible to invoke a method on a *local* object. The receiver reference's global component ι_a must match the identity of the current actor.

- FIELD-ACCESS, FIELD-UPDATE: a field update modifies the actor's heap such that it contains an object with the same address but with an updated set of fields. Again, field access and field update apply only to objects *local* to the executing actor.

- MAKE-FUTURE: a new future-resolver pair is created such that the future has an empty queue and is unresolved (its value is ϵ), and the resolver contains the future's identity ι_f. The subexpression e is further reduced with x_f and x_r bound to references to the new future and resolver respectively.

- LOCAL-ASYNCHRONOUS-SEND: an asynchronous message sent to a *local* object (i.e. an object owned by the actor executing the message send) adds a new message to the end of the actor's own message queue. The message send immediately reduces to `null`.

- PROCESS-MESSAGE: this rule describes the processing of incoming asynchronous messages directed at local objects or resolvers (but not futures). A new message can be processed only if two conditions are satisfied: the actor's queue Q is not empty, and its current expression cannot be reduced any further (the expression is a value v). The auxiliary function $process$ (see Figure 9) distinguishes between:

  - a regular message m (or the meta-level message $\mathrm{resolve}_\mu$), which is processed by invoking the corresponding method on the receiver object.
  - a two-way message m_f, as generated by the desugaring of $e \leftarrow_f m(\overline{e})$. Such a message is processed by invoking the corresponding method on the receiver object, and by sending the result of the invocation to the "hidden" last parameter r which denotes a resolver object.
  - a meta-level message $\mathrm{register}_\mu$, which indicates the registration of a callback function v, to be applied to the value of a resolved future. Since $process$ is only applicable on non-future values $\iota_a.\iota_o$, the callback function v can be triggered immediately, by asynchronously applying it to $\iota_a.\iota_o$. This ensures that v is applied later in its own turn.

- PROCESS-MSG-TO-FUTURE: this rule describes the processing of incoming asynchronous messages directed at local futures. The processing of the message depends on the state of the recipient future, as determined by the auxiliary function $store$. This function returns a tuple (m, e) where m denotes either a message or the empty sequence, and e denotes either an asynchronous message send or

`null`. The message `m` is then appended to the future's queue, and the actor continues reducing the expression e. The $store$ function determines whether to store or forward the message `m`, depending on the state of the future and the type of message:

  - If the future is unresolved (i.e. its value is still ϵ), the message is enqueued and must not be forwarded yet (e is `null`).
  - If the future is resolved and the message name m is not register_μ, the message need not be enqueued (`m` is $\emptyset$), but is rather immediately forwarded to the resolved value v.
  - If the future is resolved and the message is register_μ, which indicates a request to register a callback function $\iota_a.\iota_o$ with the future, the function is asynchronously applied to the resolved value v. This request need not be enqueued (`m` is $\emptyset$).

- RESOLVE: this rule describes the reduction of the meta-level message resolve_μ, as used in the desugaring of the "when" and "resolve" expressions. This message can only be reduced when directed at a resolver object ι_r whose paired future ι_f is still unresolved (i.e. its value is still ϵ). The paired future is updated such that it is resolved with the value v, and its queue is emptied. The messages previously stored in its queue Q' are forwarded as a sequence of message sends, as described by the auxiliary function fwd:

  - If the queue is empty, no more messages need to be forwarded and the expression reduces to `null`.
  - If the queue contains a message m or the meta-level message resolve_μ, that message is forwarded to v.
  - If the queue contains the meta-level message register_μ, this indicates a request to notify the callback function $\iota_a.\iota_o$ when the future becomes resolved. The function is thus asynchronously applied with the future's resolved value v.

Actor-global reductions We summarize the actor-global reduction rules in Figure 8:

- NEW-ACTOR: when an actor ι_a reduces an actor literal expression, a new actor $\iota_{a'}$ is added to the configuration. The new actor's heap consists of a single new object ι_o whose fields and methods are described by the literal expression. As in the rule for NEW-OBJECT, the object's fields are initialized to `null`. The new actor has an empty queue and will, as its first action, initialize the fields of its only object. The actor literal expression itself reduces to a far reference to the new object, allowing the creator actor to communicate further with the new actor.

- FAR-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send directed at a far reference, i.e. a reference whose global component $\iota_{a'}$ differs from that of the current actor ι_a. A new message is appended to the queue of the recipient actor $\iota_{a'}$. The arguments $\overline{v}$ of the message send expression are parameter-passed as described by the auxiliary function $pass$ (see Figure 9). This function prescribes the set O'' of copied isolate objects to be added to the recipient's heap and a sequence of values $\overline{v'}$ with updated addresses referring to

(NEW-ACTOR)

$$\frac{\iota_{a'}, \iota_o \text{ fresh} \qquad r = \iota_{a'}.\iota_o \qquad a' = \mathcal{A}\langle \iota_{a'}, \mathcal{O}\langle \iota_o, \mathrm{o}, \overline{f := \mathrm{null}}, \overline{m(\overline{x})\{e'\}}\rangle, \emptyset, \overline{r.f := [r/\mathrm{self}]e}\rangle}{K \uplus \mathcal{A}\langle \iota_a, O, Q, e_\Box[\mathrm{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e'\}}\}]\rangle \rightarrow_k K \cup \mathcal{A}\langle \iota_a, O, Q, e_\Box[r]\rangle \cup a'}$$

(FAR-ASYNCHRONOUS-SEND)

$$\frac{K = K' \uplus \mathcal{A}\langle \iota_{a'}, O', Q', e'\rangle \qquad (O'', \overline{v}') = pass(\iota_a, O, \overline{v}, \iota_{a'}) \qquad Q'' = \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}'\rangle \cdot Q'}{K \uplus \mathcal{A}\langle \iota_a, O, Q, e_\Box[\iota_{a'}.\iota_o \leftarrow m(\overline{v})]\rangle \rightarrow_k K' \cup \mathcal{A}\langle \iota_a, O, Q, e_\Box[\mathrm{null}]\rangle \cup \mathcal{A}\langle \iota_{a'}, O' \cup O'', Q'', e'\rangle}$$

(CONGRUENCE)

$$\frac{a \rightarrow_a a'}{K \uplus \{a\} \rightarrow_k K \cup \{a'\}}$$

Figure 8: Actor-global reduction rules.

the copied isolates, if any. As in the LOCAL-ASYNCHRONOUS-SEND rule, the message send expression evaluates to `null`.

- CONGRUENCE: this rule merely relates the local and global reduction rules.

An AT-LITE program e is reduced in an initial configuration containing a single "main" actor $K_{init} = \{\mathcal{A}\langle \iota_a, \emptyset, \emptyset, [\mathrm{null}/\mathrm{self}]e\rangle\}$. At top-level, the `self`-pseudovariable is bound to `null`.

7.3.4 Parameter-passing rules

The auxiliary function $pass(\iota_a, O, \overline{v}, \iota'_a)$ (see Figure 9) describes the rules for parameter-passing the values $\overline{v}$ from actor ι_a to actor ι'_a, where O is the heap of the sender actor ι_a.

The parameter-passing rules for AT-LITE values are simple: objects are passed by far reference, isolates are passed by copy, and `null` is passed by value. When an isolate is passed by copy, all of its constituent field values are recursively parameter-passed as well.

The auxiliary function $reach(O, \overline{v})$ returns the set of all isolate objects reachable via other isolates in O, starting from the root values $\overline{v}$. The first two cases define the terminal conditions of this traversal. In the third case, an isolate object o is encountered and added to the result. All of o's field values are added to the set of roots, and o itself is removed from the set of objects to consider, so that it is never visited twice. The fourth rule skips all other values and applies when v is `null`, a far reference $\iota_{a'}.\iota_{o'}$, an object that was already visited ($v = \iota_a.\iota_o, \iota_o \notin O$) or a non-isolate object ($v = \iota_a.\iota_o, \mathcal{O}\langle \iota_o, \mathrm{o}, F, M\rangle \in O$).

The mapping σ prescribes fresh identities for each isolate in O'. The function $pass$ prescribes the set of isolates O'_σ which is simply the set O' with all isolates renamed according to σ. The function σ_v replaces references to parameter-passed isolates with references to the fresh copies, and is the identity function for all other values.

$$
\begin{array}{rcll}
reach(\emptyset, \overline{v}) & \stackrel{def}{=} & \emptyset & \\
reach(O, \emptyset) & \stackrel{def}{=} & \emptyset & \\
reach(O \uplus o, \overline{v} \cdot \iota_a.\iota_o) & \stackrel{def}{=} & reach(O, \overline{v} \cdot \overline{v'}) \cup \{o\} & \text{if } o = \mathcal{O}\langle \iota_o, \text{I}, \overline{f := v'}, M\rangle \\
reach(O, \overline{v} \cdot v) & \stackrel{def}{=} & reach(O, \overline{v}) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
pass(\iota_a, O, \overline{v}, \iota'_a) & \stackrel{def}{=} & (O'_\sigma, \sigma_v\, \overline{v}) \\
 & & \text{where } O' = reach(O, \overline{v}) \\
 & & \sigma = \{\iota_o \mapsto \iota'_o \mid \mathcal{O}\langle \iota_o, t, F, M\rangle \in O', \iota'_o \text{ fresh }\} \\
 & & O'_\sigma = \{\mathcal{O}\langle \sigma(\iota_o), \text{I}, \overline{f := \sigma_v(v)}, M\rangle \mid \mathcal{O}\langle \iota_o, \text{I}, \overline{f := v}, M\rangle \in O'\} \\
 & & \sigma_v(v) = \begin{cases} \iota'_a.\iota'_o & \text{if } v = \iota_a.\iota_o,\ \iota_o \mapsto \iota'_o \in \sigma \\ v & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rcll}
store(m, \overline{v}, \epsilon) & \stackrel{def}{=} & (\mathcal{M}\langle \epsilon, m, \overline{v}\rangle, \text{null}) & \\
store(m, \overline{v}, v) & \stackrel{def}{=} & (\emptyset, v \leftarrow m(\overline{v})) & m \neq \text{register}_\mu, v \neq \epsilon \\
store(m, \iota_a.\iota_o, v) & \stackrel{def}{=} & (\emptyset, \iota_a.\iota_o \leftarrow \text{apply}(v)) & m = \text{register}_\mu, v \neq \epsilon
\end{array}
$$

$$
\begin{array}{rcll}
fwd(v, \emptyset) & \stackrel{def}{=} & \text{null} & \\
fwd(v, Q \cdot \mathcal{M}\langle \epsilon, m, \overline{v}\rangle) & \stackrel{def}{=} & v \leftarrow m(\overline{v})\ ;\ fwd(v, Q) & m \neq \text{register}_\mu \\
fwd(v, Q \cdot \mathcal{M}\langle \epsilon, m, \iota_a.\iota_o\rangle) & \stackrel{def}{=} & \iota_a.\iota_o \leftarrow \text{apply}(v)\ ;\ fwd(v, Q) & m = \text{register}_\mu
\end{array}
$$

$$
\begin{array}{rcll}
process(\iota_a.\iota_o, m, \overline{v}) & \stackrel{def}{=} & \iota_a.\iota_o.m(\overline{v}) & m \neq m_f, m \neq \text{register}_\mu \\
process(\iota_a.\iota_o, m_f, \overline{v} \cdot r) & \stackrel{def}{=} & r \leftarrow \text{resolve}_\mu(\iota_a.\iota_o.m(\overline{v})) & \\
process(\iota_a.\iota_o, \text{register}_\mu, v) & \stackrel{def}{=} & v \leftarrow \text{apply}(\iota_a.\iota_o) &
\end{array}
$$

Figure 9: Auxiliary functions used in the reduction rules.

7.4 Service Discovery

In Figure 10, we extend AT-LITE with primitives for service discovery, allowing objects in different actors to discover one another as described in Section 6.1.

AT-LITE actors are extended with a set of exported objects E and a set of import callbacks I. Values are extended to include type tags τ. Objects can be exported, and callbacks can be registered, under various type tags. When the tags match, the callback is fired. The AT-LITE syntax is extended with tag literals and expressions to export objects, to register callbacks for discovery and the syntactic sugar $\text{whenDiscovered}(e \rightarrow x)\{e'\}$ to resemble the AmbientTalk `when:discovered:` function.

Figure 11 lists the additional reduction rules for service discovery:

- PUBLISH: to reduce an `export` expression, the first argument must be reduced to a type tag τ and the second argument must be reduced to a reference (which may be a far reference). The effect of reducing an `export` expression is that the

Semantic Entities

$$a \in A \subseteq \mathbf{Actor} \quad ::= \quad \mathcal{A}\langle \iota_a, O, Q, E, I, e\rangle$$
$$v \in \mathbf{Value} \quad ::= \quad \ldots \mid \tau$$
$$\tau \in \mathbf{Type}$$

Syntax

$$e \quad ::= \quad \ldots \mid \tau \mid \text{export } e\ e \mid \text{discover } e\ e \mid \text{whenDiscovered}(e \to x)\{e\}$$

Evaluation Contexts

$$e_\Box \quad ::= \quad \ldots \mid \text{export } e_\Box\ e \mid \text{export } v\ e_\Box \mid \text{discover } e_\Box\ e \mid \text{discover } v\ e_\Box$$

Syntactic Sugar

$$\text{whenDiscovered}(e \to x)\{e'\} \quad \stackrel{\text{def}}{=} \quad \text{discover } e\ (\lambda x.e')$$

Figure 10: Extensions for Service Discovery.

(PUBLISH)

$$\frac{(O', v') = pass(\iota_a, O, \iota_{a'}.\iota_o, \iota_a)}{\mathcal{A}\langle \iota_a, O, Q, E, I, e_\Box[\text{export } \tau\ \iota_{a'}.\iota_o]\rangle \to_a \mathcal{A}\langle \iota_a, O, Q, E \cup (O', v', \tau), I, e_\Box[\text{null}]\rangle}$$

(SUBSCRIBE)

$$\mathcal{A}\langle \iota_a, O, Q, E, I, e_\Box[\text{discover } \tau\ \iota_a.\iota_o)]\rangle \to_a \mathcal{A}\langle \iota_a, O, Q, E, I \cup (\iota_a.\iota_o, \tau), e_\Box[\text{null}]\rangle$$

(MATCH)

$$\frac{\mathcal{A}\langle \iota_{a'}, O', Q', E' \uplus (O'', v, \tau), I', e'\rangle \in K \qquad (O''', v') = pass(\iota_{a'}, O'', v, \iota_a) \qquad Q'' = \mathcal{M}\langle \iota_a.\iota_o, \text{apply}, v'\rangle \cdot Q}{K \uplus \mathcal{A}\langle \iota_a, O, Q, E, I \uplus (\iota_a.\iota_o, \tau), e\rangle \to_k K \cup \mathcal{A}\langle \iota_a, O \cup O''', Q'', E, I, e\rangle}$$

Figure 11: Reduction rules for service discovery.

actor's set of exported objects E is extended to include the exported object and tag. An exported object is parameter-passed as if it were included in an inter-actor message. Hence, if the object is an isolate, a copy of the isolate is made at the time it is exported.

- SUBSCRIBE: to reduce a `discover` expression, the first argument must be reduced to a type tag τ and the second argument must be reduced to an object reference. The effect of reducing a `discover` expression is that the actor's set of import callbacks I is extended to include a reference to the local callback object, and the tag.

- MATCH: this rule is applicable when a configuration of actors contains both an actor $\iota_{a'}$ that exports an object under a type tag τ, and a different actor ι_a that has registered a callback under the same tag τ. The effect of service discovery is that an asynchronous `apply` message will be sent to the callback object in ι_a. The callback is simultaneously removed from the import set of its actor so that it can be notified at most once. The exported object v is parameter-passed again, this time to copy it from the publication actor $\iota_{a'}$ to the subscription actor ι_a.

7.5 Robust time-decoupled message sends

So far, we have assumed that AT-LITE actors are always permanently connected to all other actors. In practice, actors may reside on distributed devices and only occasionally connect to deliver messages. In Figure 12, we extend AT-LITE actors with *networks*. Networks partition actors such that communication between actors is only possible if they are in the same network. A network is represented as a unique identifier.

Semantic Entities

$$
\begin{array}{rcl}
a \in A \subseteq \mathbf{Actor} & ::= & \mathcal{A}\langle \iota_a, O, Q, Q_{out}, \iota_n, e \rangle \\
Q_{out} \in \mathbf{Outbox} & ::= & \iota_a \mapsto \bar{l} \\
l \in \mathbf{Envelope} & ::= & (\mathrm{m}, O_{\mathrm{m}}) \\
\iota_n \in \mathbf{NetworkId} & &
\end{array}
$$

Figure 12: Extensions for time-decoupled message sends.

The use of networks allows us to more faithfully describe AmbientTalk's remote message passing semantics with buffering of messages sent to far references (see Section 6.2). Asynchronous message sends are now split into two parts: message creation and message transmission. Whenever an actor reduces the $\leftarrow$ operator, a message is created and stored in a message outbox (called Q_{out}), to be transmitted when the recipient is connected. This is called *time-decoupled communication* [EFGK03], as actors do not require an active network connection at the time they send a message to each other.

We represent an actor's outbox Q_{out} as a function that, for each remote actor ι_a, stores all outgoing messages addressed to objects owned by ι_a. The outgoing messages $Q_{out}(\iota_a)$ are represented as an ordered sequence of envelopes $\bar{l}$. An envelope is simply a message m together with the set of isolate objects O_{m} passed as arguments to that message. These objects have to be passed together with the message upon transmission.

In the reduction rules, the original rule for FAR-ASYNCHRONOUS-SEND is replaced by new rules for message creation (CREATE-MESSAGE) and message transmission (TRANSMIT-MESSAGE). Figure 13 lists the additional reduction rules for time-decoupled message transmission:

- CREATE-MESSAGE: This rule creates a new envelope and appends it to $Q_{out}(\iota_{a'})$, i.e. the list of outgoing messages addressed at actor $\iota_{a'}$. This rule is actor-local, so it is applicable regardless of whether the recipient actor is currently in the same network.

- TRANSMIT-MESSAGE: This rule is applicable whenever an actor is in the same network as an actor for which it has undelivered messages. If this is the case, the last (i.e. eldest) of these undelivered messages is removed from the sender actor's outbox and appended to the destination actor's inbound message queue.

- MOBILITY: This rule describes that actors can switch between different networks. Application of this rule is entirely involuntary, i.e. actors do not themselves choose to move, they are moved around (non-deterministically) by the system or environment.

(FAR-ASYNCHRONOUS-SEND)
This rule is removed.

(CREATE-MESSAGE)
$$\frac{(O_{\mathrm{m}}, \overline{v}') = pass(\iota_a, O, \overline{v}, \iota_{a'}) \qquad \mathrm{m} = \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}' \rangle \quad \bar{l} = Q_{out}(\iota_{a'}) \qquad Q'_{out} = Q_{out}[\iota_{a'} \mapsto (\mathrm{m}, O_{\mathrm{m}}) \cdot \bar{l}]}{\mathcal{A}\langle \iota_a, O, Q, Q_{out}, \iota_n, e_{\Box}[\iota_{a'}.\iota_o \leftarrow m(\overline{v})] \rangle \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, Q'_{out}, \iota_n, e_{\Box}[\mathrm{null}] \rangle}$$

(TRANSMIT-MESSAGE)
$$\frac{Q_{out}(\iota_{a'}) = \bar{l} \cdot (\mathrm{m}, O_{\mathrm{m}}) \qquad K = K' \cup \mathcal{A}\langle \iota_{a'}, O', Q', Q'_{out}, \iota_n, e' \rangle}{K \cup \mathcal{A}\langle \iota_a, O, Q, Q_{out}, \iota_n, e \rangle \rightarrow_k K' \cup \mathcal{A}\langle \iota_a, O, Q, Q_{out}[\iota_{a'} \mapsto \bar{l}], \iota_n, e \rangle \cup \mathcal{A}\langle \iota_{a'}, O' \cup O_{\mathrm{m}}, \mathrm{m} \cdot Q', Q'_{out}, \iota_n, e' \rangle}$$

(MOBILITY)
$$\frac{\iota_{n'} \in \mathbf{NetworkId} \qquad \iota_n \neq \iota_{n'}}{K \cup \mathcal{A}\langle \iota_a, O, Q, Q_{out}, \iota_n, e \rangle \rightarrow_k K \cup \mathcal{A}\langle \iota_a, O, Q, Q_{out}, \iota_{n'}, e \rangle}$$

Figure 13: Reduction rules for time-decoupled message sends.

7.6 Semantic Properties

In Section 5.5 we argued that AmbientTalk's event loop concurrency model avoids low-level data races and deadlocks by design. While we do not formally prove these properties for AT-LITE, we argue that they hold based on the reduction rules. First, data races are prevented by partitioning objects among different actors, and by restricting field updates (FIELD-UPDATE) to be applicable only to *local* objects. Second, deadlocks are prevented by not providing any synchronization primitive that can block an actor while reducing a compound expression. Actors can only receive external messages when their expression has been fully reduced to a value (PROCESS-MESSAGE,PROCESS-MSG-TO-FUTURE). Of course, it remains possible to write AT-LITE programs that lead to other forms of lost progress, e.g. by not resolving a future that has callbacks awaiting its value.

8 Implementation

An AmbientTalk interpreter written in Java is available. Initially, we targeted the Java 2 micro edition (J2ME) platform, under the connected device configuration (CDC). In 2006, this allowed AmbientTalk to run on PDAs (the forerunners of contemporary smartphones). Today, we primarily target the Android OS, which supports Java programs via the Dalvik Virtual Machine. Our current experimental setup is a combination of Samsung Nexus S and Galaxy Nexus phones (all running Android 4.0.4) and Motorola Xoom tablets (running Android 3.2). The AmbientTalk interpreter is published on the Google Play Store[1]. It runs on devices featuring Android 2.1 or higher, which makes it compatible with 99% of Android devices at the time of writing.

[1]The interpreter can be downloaded at `http://bit.ly/HM7Kzv`.

AmbientTalk interpreters in the same ad hoc network communicate with one another by means of standard TCP/IP. AmbientTalk's topic-based publish/subscribe service discovery mechanism is peer-to-peer and does not require a centralised repository. AmbientTalk interpreters discover one another by means of the network's support for multicast messaging using UDP. After a successful discovery, the two interpreters exchange discovery information (e.g. registered subscriptions and exported objects) in order to find a match.

As described previously, the naming and discovery of services happens via type tags. We make the underlying assumption that the name of such tags is known by all participating services. This discovery mechanism also does not take versioning into account explicitly, e.g. if a certain service is updated, older clients may discover the updated service, and clients that want to use only the updated service may still discover older versions. Clients and services are thus themselves responsible to check versioning constraints.

9 Language Extensions

An explicit design goal of AmbientTalk was to serve as a research language in which the design space of mobile networks and actor languages could be explored. Here, we provide a brief overview of the various language extensions developed in and for AmbientTalk in the past six years. First, extensions geared towards mobile networks:

Ambient references Ambient references are a new type of far reference that can automatically rebind to different objects during their lifetime [Van08]. As such, they support *roaming* in mobile applications. An ambient reference is initialized with a type tag, and can, during its lifetime, refer to *any* object exported with that tag. It was inspired by M2MI's omnihandles [KB02].

RFID tags AmbientTalk has been extended to program *mobile RFID-enabled applications* [LPD11]. These applications use RFID tags to represent physical objects. Just like wireless communication in mobile networks, communicating with RFID tags is prone to many failures. We allow AmbientTalk objects to be serialized on RFID tags, and allow applications to communicate with these stored objects using a special type of far reference that makes abstraction of low-level RFID communication details and similarly provides a smart buffering to mask intermittent communication failures.

Network-aware references Modern smartphones support multiple communication technologies, such as Wi-Fi, Bluetooth, cellular data and possibly near field communication (NFC). Most of these can be used to form ad hoc peer-to-peer networks. AmbientTalk supports multiple such technologies via different communication drivers. This raises issues for programmers: if a program is connected to various overlapping networks, objects can be discovered multiple times, and each of these far references can disconnect or reconnect individually. *Network-aware references* [PHD11] are a new type of far reference that group different such individual far references so that programmers can make abstraction from the underlying technology.

Tuple space-based abstractions Tuple spaces are a coordination mechanism in which processes communicate by means of a shared associative memory (the tuple

space), originating in the Linda coordination language [Gel85]. Communication via tuples forms an alternative to AmbientTalk's message passing abstractions. TOTAM ("Tuples on the ambient") [SBM09] is a tuple space library for AmbientTalk in which tuples can propagate through a mobile network, inspired by a similar model named TOTA [MZ04]. A TOTAM tuple can contain rules that describe the conditions under which it is visible to peer applications, facilitating the development of context-sensitive applications [SGBDMD10].

Second, extensions geared towards actor languages in general:

Parallel actor monitors While the actor model is well appreciated for its ease of use, its scalability is often criticized. Indeed, the fact that execution *within* an actor is sequential prevents truly parallel reads on state encapsulated by that actor. To address this issue, AmbientTalk has been extended with Parallel Actor Monitors (PAM) [STD10]. A PAM is a modular, reusable scheduler that permits to introduce parallelism within an actor in a local and abstract manner. PAM allows the stepwise refinement of local parallelism within a system on a per-actor basis, without having to deal with low-level synchronization details and locking.

Reactive Programming AmbientTalk applications are inherently event-driven. First because of services that dynamically and unpredictably disconnect and reconnect, and second because of the language's event loop concurrency model. This event model has to integrate with events originating from other sources, e.g. the UI or sensors of the mobile device (GPS, RFID). AmbientTalk's reliance on traditional event handlers gives rise to the well-known problem of *inversion of control* [HO06], making complex event-driven code hard to manage. AmbientTalk/R [LMVD10] is an extension of AmbientTalk that supports reactive programming, a programming paradigm that models events as state changes in time-varying values. Code that depends on such time-varying values is automatically tracked by the interpreter and re-executed whenever these values change. Experiments have been conducted with a visual language on top of AmbientTalk/R to graphically edit distributed reactive programs [LD10].

Contracts AmbientTalk has been extended with higher-order contracts, modelled after [FF02]. Contracts can specify pre- and post-conditions on functions. Aside from performing simple type checks on function arguments and return values, AmbientTalk's contract system additionally enables one to express constraints on what a function can do during its execution (e.g. a contract may state that the function is not allowed to send remote messages). While contracts were initially designed for sequential function calls, in AmbientTalk they are also applicable to asynchronous message sends between remote objects [SHT+11].

10 Applications and Experiments

Over the past few years, we have written several applications in AmbientTalk, to showcase the language's suitability to build a variety of mobile peer-to-peer applications:

- A P2P drawing application named *weScribble* in which participants can collaboratively draw on a shared virtual canvas. This application is released on the Android Market.

- A P2P chat application in which participants automatically discover one another. Messages sent to disconnected peers are automatically buffered and sent upon reconnection.
- A P2P "flea market" application (based on an example by Eugster *et al.* [EGH05]) in which sellers can post items for sale, and buyers can indicate their interest in general item categories. When a buyer and seller connect, the application exchanges contact details of both parties.
- A P2P music match maker: peers can upload a list of song titles. When peers meet, they exchange and compare song titles. If both users share a similar taste in music, they are notified. If a user disconnects during the exchange, the exchange is automatically stalled until both peers reconnect. For this application, we established a detailed comparison in terms of lines of code with an equivalent application in Java [2].
- A P2P social networking application, supporting plug-in applications (inspired by the Facebook platform). One example application is a picture sharing app that allows users to share recently taken pictures with people in their direct proximity.
- Various multiplayer games, including a P2P version of the classic Atari game Pong, a P2P Rock-Paper-Scissors game, a P2P scrabble-like game, and a so-called "urban game" named *Flikken* [SGBDMD10] to be played outdoors, where players are either "cops" or "thieves" and where interaction depends on the players' GPS location.
- A workflow language named NOW in which one can express workflow patterns for nomadic networks [PVDSJ10].

In addition, AmbientTalk is actively used as a teaching platform in the Master in Computer Science curriculum at the Vrije Universiteit Brussel, for a course on mobile and distributed computing. Students use AmbientTalk for simple exercises, as well as more advanced projects such as implementing distributed hash tables or multi-player games.

11 Asynchronous and Distributed Debugging

To support the development of mobile applications in AmbientTalk, we have built a number of tools. Our most significant effort thus far is *IdeAT*, an AmbientTalk IDE developed as a plugin for Eclipse [3]. IdeAT provides many of the standard features expected by modern IDEs, such as a source code editor, a launcher to run AmbientTalk programs with an interactive console, and a debugger. The AmbientTalk editor supports syntax highlighting, syntax error reporting, brace matching, auto-format, auto-completion of the current identifier and code completion for control structures such as `if:then:else:`.

The IdeAT debugger is called REME-D (pronounce "remedy"), the Reflective Epidemic MEssage-oriented Debugger [GBCVC+11]. REME-D is an online debugger

[2] The results of this experiment are available at `http://ambienttalk.googlecode.com/files/LeasingExperimentJavaRMIvsAmbientTalk.zip`.

[3] The IdeAT plugin is available from the Eclipse update site at `http://tinyurl.com/ideat`

designed to debug mobile, distributed applications, and specifically to debug event-driven programs. REME-D was not designed to debug sequential AmbientTalk code, but only the asynchronous, distributed part of applications.

REME-D allows one to inspect the message queue of an actor, to set breakpoints on incoming asynchronous messages and to step through the program at the granularity of whole messages rather than individual statements. In addition, inspired by Causeway [SCM09], REME-D incorporates features from post-mortem, message-oriented debuggers, such as logging all sent and received messages. This allows the programmer to browse the causal history of message events.

REME-D has one additional feature to support the deployment of an application to be debugged on a mobile ad hoc network: *epidemic debugging*. Epidemic debugging allows a peer program that is discovered in the network to join an ongoing debugging session at runtime. If that program supports REME-D, it can be switched into debug-mode and start "infecting" other programs that it encounters, so that they too enter debug-mode. Thus, a federation of mobile programs can easily take part in a single distributed debugging session, without having to explicitly configure each program as a participant ahead of time. Programs can leave a debugging session at any time without disrupting the debug-mode of remaining participants, either by disconnecting or in response to user action.

Figure 14 shows a REME-D debugging session of an online shopping application (the example was taken from [SCM09]). Three views are on display: the debug view on the top right, the state inspector on the top left, and a source code editor on the bottom. In the debug view, we can see that the application is actually distributed and consists of two programs, one running `Store.at` (a shop) and the other running `Buyer.at` (a client of the shop). Each of these programs consists of multiple actors, as indicated by the nested actor ID's.

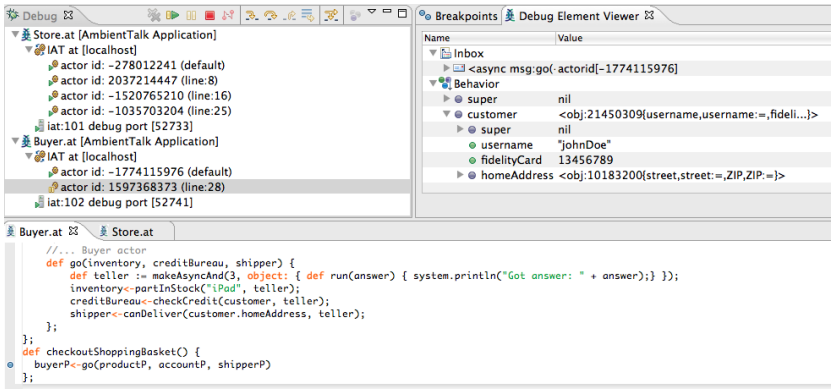

Figure 14: AmbientTalk IDE for Eclipse showing a REME-D debug session.

The editor shows part of the implementation of the `Buyer.at` application. The `checkoutShoppingBasket` method is called when the customer purchases a product. In response to this action, a `go` message is sent, which verifies three things before accepting the product order: 1) whether the requested item is still in stock, 2) whether the customer payment is valid and 3) whether the order can be delivered in time.

Finally, the state inspector gives us a view on the state of the objects owned by the actors, as well as the messages that await processing on the incoming message queue.

The actor highlighted in this example contains `customer` and `shoppingCart` objects, and has a `go` message pending in its incoming message queue (inbox). As can be seen in the screenshot, the user can use the inspector to navigate the actor's entire heap of objects reachable from its top-level fields.

A user can place breakpoints in the source code at places where he or she wants to inspect the application state. In AmbientTalk, this is often at places where asynchronous messages are sent or received. In Figure 14, a breakpoint is set on the outgoing `go` asynchronous message, as indicated by the blue dot in the editor's left margin. What is unique about such breakpoints in REME-D is that execution of the *receiver* actor will be paused when the breakpointed message reaches the head of its message queue, *before* the message is processed. The figure shows an actor of the `Buyer.at` program paused as a result of this breakpoint. As shown by the state inspector on the top right, the `go` message is still in the message queue.

In REME-D, one can step through an application one entire event loop *turn* at a time. As in regular breakpoint-based debuggers, three kinds of step commands are offered: step-over, step-into and step-return a turn, each explained below.

A *step-over* command instructs the debugger to let an actor process a single message, the one at the head of its queue, and return it to the paused state. By stepping over a turn, one can observe how the state of the actor changes as it processes incoming messages.

A *step-into* command instructs the debugger to let an actor process a single message (in a single turn), and return it to the paused state. In addition, all actors that *received* messages sent during that turn are *also* paused. The user can then verify the actor's state, and decide which of the now paused recipient actors he or she wants to continue debugging.

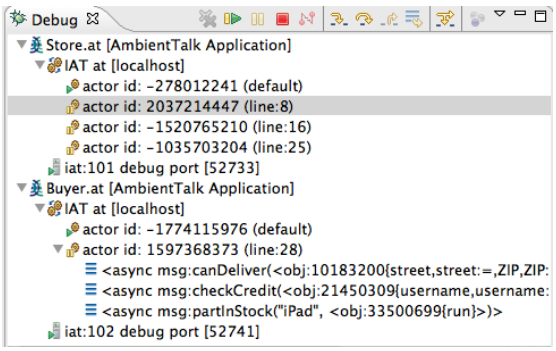

Figure 15: Debug view after a step-into command.

Figure 15 shows the debug view after having stepped into the turn that processed the `go` message shown in Figure 14. The actor that processed the `go` message is expanded, which reveals the list of messages sent while processing the `go` method. Each of these messages (`canDeliver`, `checkCredit` and `partInStock`) was directed at a different actor in the `Store.at` application. As indicated by the little yellow "pause" signs, each of these three actors is now paused.

Finally, a *step-return* command instructs the debugger to return from a message which was previously stepped into. This is useful when debugging a method that was invoked by sending a two-way message `m` with an associated future `f`. The method

must then asynchronously "return" its result to `f`. When step-returning from such a method, the actor that previously *sent* the message `m` is paused, at the point where `f` becomes resolved. The user can then easily inspect the return value of the asynchronously invoked method. Furthermore, at that point, the user can continue the debugging session from the start of any callback functions registered with `f` using the `when:becomes:catch:` function (see Section 5.4). Hence, REME-D users can debug two-way asynchronous messages as easily as one would debug synchronous method invocations using a traditional debugger.

REME-D is developed in AmbientTalk itself, which is made possible by the language's support for reflection. AmbientTalk provides a set of hooks that allow programs to customize the message reception and processing behavior of actors [MVCT+09]. REME-D makes use of these hooks to instrument actors for debugging. For details, we refer to [GBCVC+11].

12 Related Work

In Section 3, we already introduced the languages and systems that directly influenced the design of AmbientTalk. Here, we briefly highlight related work in two broad categories: languages and systems also directed at developing software for mobile networks, and related work in actor-based languages in general.

Mobile networks AmbientTalk tackles the issues of mobile networks by building on object-oriented abstractions such as object references and message-passing. Others have tackled the same issues by building on different communication paradigms. For example, LIME [MPR01] and TOTA [MZ04] are mobile computing middleware based on tuple spaces [Gel85]. In the tuple space model, processes do not communicate by sending each other private messages, but rather by inserting and removing tuples from a shared associative store (the tuple space).

Another fruitful paradigm for mobile computing is Publish/Subscribe [EFGK03]. The main difference between traditional, centralised publish/subscribe architectures and those for mobile networks is the incorporation of geographical constraints on the event disseminations and subscriptions. For example, in location-based Publish/Subscribe (LPS) [EGH05] and STEAM [MCNC05], publishers and subscribers can define a geographical range to scope their publications or subscriptions. Only when the ranges overlap is an event disseminated to the subscriber. AmbientTalk's service discovery mechanism is based on the publish/subscribe paradigm, but does not provide any explicit means to scope exported objects and subscribed event handlers.

Actor-based systems In the original actor model, actors refer to one another via *mail addresses* [Agh86]. When an actor sends a message to a recipient actor, the message is placed in a mail queue and is guaranteed to be eventually delivered by the actor system. Most practical implementations of the actor model do not actually guarantee eventual delivery.

For instance, in the E language [MTS05], a network disconnection immediately *breaks* a far reference. Once the reference is broken, it will no longer deliver any messages. Hence, E's far references do not try to mask intermittent network failures the way AmbientTalk far references do. When leased far references are used, AmbientTalk does not guarantee eventual delivery of messages either, as these references may expire (cf. Section 6.2).

The fact that actors can only communicate asynchronously makes the original actor model by itself almost suitable for mobile networks. However, the actor model lacks a means to perform service discovery, i.e. to acquire the mail address of an unknown remote actor without a common third party acting as an introducer. There do exist extensions of the actor model that tackle this issue. In the ActorSpace model [CA94], messages can be sent to a *pattern* rather than to a mail address, and they will be delivered by the actor system to an actor with a matching pattern. The ActorSpace model, however, was not designed for mobile networks, as it relies on infrastructure to manage the matching of the patterns.

Futures (also known as promises) are a frequently recurring abstraction in actor systems. The use of futures as return values from asynchronous message sends can be traced back to actor languages such as ABCL/1 [YBS86]. In Argus, promises additionally supported pipelined message sends and exceptions [LS88]. Most future abstractions support synchronisation by suspending a thread that accesses an unresolved future. This style of synchronization is called *wait-by-necessity* [Car93]. The E language pioneered the `when`-expression to await the value of a promise in a non-blocking way [MTS05]. In other actor systems, the same goal is often accomplished by passing explicit callbacks or "continuation" actors as arguments to a message.

Our notion of future-resolver pairs descends directly from promise-resolver pairs in E, which are themselves inspired by logic variables in concurrent constraint programming [Sar93].

The view of AmbientTalk actors as containers of regular objects is based on E's similar notion of actors as *vats* [MTS05]. In JCobox [SPH10], actors are similarly represented as *coboxes*. JCobox additionally supports cooperative multitasking (coroutines) within a cobox.

13 Conclusion

Developing responsive peer-to-peer mobile applications is a challenge because of the inherent characteristics of mobile networks. Devices are only sporadically connected and need to discover one another on the move, without always being able to rely on a shared infrastructure.

AmbientTalk is designed as a language to facilitate the development of mobile P2P applications. It features a built-in model for concurrency and distribution based on the actor model, a model that was developed for open, asynchronous, distributed systems. AmbientTalk augments and extends this model with built-in support for service discovery and fault-tolerant, time-decoupled asynchronous message sending. We have given both an informal and a formal account of these core language features. To the best of our knowledge, this is the first formal account of an actor language built on the communicating event loops model.

References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[BBC+06] Laurent Baduel, Franc$_c$oise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[Bri88] J.-P. Briot. From objects to actors: study of a limited symbiosis in smalltalk-80. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 69–72, New York, NY, USA, 1988. ACM Press.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–343, 2004.

[CA94] C. J. Callsen and G. Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, 1994.

[Car93] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[DVM+06] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[EFGK03] P. Th. Eugster, Pascal A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Survey*, 35(2):114–131, 2003.

[EGH05] P.Th. Eugster, B. Garbinato, and A. Holzer. Location-based publish/-subscribe. *Fourth IEEE International Symposium on Network Computing and Applications*, pages 279–282, 2005.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[FFF09] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[GBCVC+11] E. Gonzalez Boix, Noguera C., T. Van Cutsem, W. De Meuter, and T. D'Hondt. REME-D: a Reflective, Epidemic Message-Oriented Debugger for Ambient-Oriented Applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), Taichung, Taiwan, March 21–25, 2011*, volume 2, pages 1275–1281. ACM, 2011.

[GC89] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 202–210, New York, NY, USA, 1989. ACM Press.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[HO07] Philipp Haller and Martin Odersky. Actors that unify threads and events. In Amy L. Murphy and Jan Vitek, editors, *Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4467 of *Lecture Notes in Computer Science*, pages 171–190. Springer, 2007.

[JdT+95] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 156–171, Colorado, December 1995.

[KB02] Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 72–73. ACM Press, 2002.

[LD10] Andoni Lombide Carreton and Theo D'Hondt. A hybrid visual dataflow language for coordination in mobile ad hoc networks. In Dave Clarke and Gul A. Agha, editors, *Coordination Models and Languages, 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6116, pages 76–91. Springer, 2010.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[Lie87] Henry Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[LMVD10] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141, pages 41–60. Springer, 2010.

[LPD11] Andoni Lombide Carreton, Kevin Pinte, and Wolfgang De Meuter. Software abstractions for mobile rfid-enabled applications. *Software: Practice and Experience*, 2011.

[LS88] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[McA95] Jeff McAffer. Meta-level programming with coda. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 190–214, London, UK, 1995. Springer-Verlag.

[MCNC05] René Meier, Vinny Cahill, Andronikos Nedos, and Siobhán Clarke. Proximity-based service discovery in mobile ad hoc networks. In *Distributed Applications and Interoperable Systems*, pages 115–129. Springer, 2005.

[MLE02] C. Mascolo, L.Capra, and W. Emmerich. Mobile Computing Middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.

[MPR01] Amy L. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.

[MTS05] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[MVCT+09] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in ambienttalk. *Softw. Pract. Exper.*, 39(7):661–699, 2009.

[MZ04] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, page 263, Washington, DC, USA, 2004. IEEE Computer Society.

[Ous96] John Ousterhout. Why threads are a bad idea (for most purposes), 1996. Presentation given at the 1996 Usenix Annual Technical Conference, January 1996. `http://www.softpanorama.org/People/Ousterhout/Threads` (captured in March 2008).

[PHD11] Kevin Pinte, Dries Harnie, and Theo D'Hondt. Enabling cross-technology mobile applications with network-aware references. In *13th International Conference on Coordination Models and Languages, COORDINATION 2011*, volume 6721 of *Lecture Notes in Computer Science*, Heidelberg, 2011. Springer-Verlag.

[PVDSJ10] Eline Philips, Ragnhild Van Der Straeten, and Viviane Jonckers. Now: a workflow language for orchestration in nomadic networks. In *Proceedings of the 12th international conference on Coordination Models and Languages*, COORDINATION'10, pages 31–45, Berlin, Heidelberg, 2010. Springer-Verlag.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. MIT Press, Cambridge, MA, USA, 1993.

[SBM09] Christophe Scholliers, Elisa Gonzalez Boix, and Wolfgang De Meuter. Totam: Scoped tuples for the ambient. *ECEASST*, 19, 2009.

[SCM09] Terry Stanley, Tyler Close, and Mark S. Miller. Causeway: A message-oriented distributed debugger. Technical Report HPL-2009-78, HP Laboratories, 2009.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003.

[SGBDMD10] C. Scholliers, E. Gonzalez Boix, W. De Meuter, and T. D'Hondt. Context-aware tuples for the ambient. *On the Move to Meaningful Internet Systems, OTM 2010*, pages 745–763, 2010.

[SHT+11] Christophe Scholliers, Dries Harnie, Éric Tanter, Wolfgang De Meuter, and Theo D'Hondt. Ambient contracts: verifying and enforcing ambient object compositions á la carte. *Personal Ubiquitous Comput.*, 15(4):341–351, April 2011.

[SJ75] Gerald Jay Sussman and Guy L Steele Jr. Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*, 1975.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[SPH10] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: generalizing active objects to concurrent components. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[STD10] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel Actor Monitors. In *14th Brazilian Symposium on Programming Languages*, September 2010.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[Van08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008.

[VCBDM09] Tom Van Cutsem, Alexandre Bergel, Stéphane Ducasse, and Wolfgang Meuter. Adding state and visibility control to traits using lexical nesting. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 220–243, Berlin, Heidelberg, 2009. Springer-Verlag.

[VMD09] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages Systems & Structures*, 35(1):80–98, April 2009.

[VMG+07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.

[Wal01] Jim Waldo. Constructing ad hoc networks. In *IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 9, 2001.

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

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