AmbientTalk: programming responsive mobile peer-to-peer applications with actors

Tom Van Cutsem[a,1,*], Elisa Gonzalez Boix[a,2], Christophe Scholliers[a,3], Andoni Lombide Carreton[a], Dries Harnie[a,2], Kevin Pinte[a], Wolfgang De Meuter[a]

[a] *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 is challenging because of problems inherent to concurrent and distributed programming, and because of problems inherent to mobile networks, such as the fact that wireless network connectivity is often intermittent, and the lack of centralized infrastructure to coordinate the peers.

We present AmbientTalk, a distributed programming language designed specifically to develop mobile peer-to-peer applications. AmbientTalk aims to make it easy to develop mobile applications that are resilient to network failures by design. We describe the language's concurrency and distribution model in detail, as it lies at the heart of AmbientTalk's support for responsive, resilient application development. The model is based on communicating event loops, itself a descendant of the actor model. We contribute a small-step operational semantics for this model and use it to establish data race and deadlock freedom.

Keywords: actors, event loops, futures, mobile networks, peer-to-peer, service discovery, leasing

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 [1]).

*Corresponding author
Email address: tvcutsem@vub.ac.be (Tom Van Cutsem)

[1] Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

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

[3] 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. 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 [2]: loosely coupled, asynchronously communicating components.

We show how AmbientTalk facilitates the development of mobile P2P applications that are resilient to intermittent network failures by default, and how this compares to mainstream distributed object-oriented middleware such as Java RMI (Section 7).

Novelty. This paper complements previous expositions of AmbientTalk [3–5] with a precise operational semantics (Section 8). To the best of our knowledge, this is the first formal account of an actor-based language based on communicating event loops with non-blocking futures. We use the operational semantics to establish data race and deadlock freedom.

Availability. AmbientTalk currently runs as an interpreter on top of the JVM and specifically targets Android-powered smartphones. It is open sourced under an MIT license and available at `ambienttalk.googlecode.com`. The Android version is published on the Google Play Store (`http://bit.ly/HM7Kzv`).

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 (e.g. Java RMI [6], Jini [7]) or programming languages (e.g. Erlang [8], Emerald [9], Argus [10]). In the spirit of such systems, we designed AmbientTalk as a new distributed 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 [11] and Smalltalk [12]. Like Ruby, however, AmbientTalk is text-based (not image-based). Inspired by Scheme [13] and E [14], it places an additional emphasis on lexical nesting of objects and lexical scoping.

Concurrency AmbientTalk embraces actor-based concurrency [2]. In particular, it embraces a particular flavor of actor-based concurrency known as *communicating event loops*, pioneered by the E programming language [14], 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 [15, 16].

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 [17], 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 [7].

Following ABCL [18], Eiffel// [19], E [14] and Argus [20], 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 [21] 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 [7], 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 [22], 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 [23] 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 [24, 25].

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 [26].

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 [23]. 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. AmbientTalk's reflective architecture is described in full detail elsewhere [27].
- Object composition. AmbientTalk features inheritance among objects (as in Self). It also supports traits [28], a more robust alternative to multiple inheritance. AmbientTalk traits are described in full detail elsewhere [29].

5. Concurrent AmbientTalk

As mentioned previously, AmbientTalk's concurrency model is based on actors [2]. 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 [14]. What sets this model apart from most other actor languages (such as Act1 [30], ABCL [18], Actalk [31], Salsa [32], Erlang [8], Kilim [33], ProActive [34] or Scala actors [35]), 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 (we discuss how method results are handled in Section 5.4). 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 synchronized (i.e. the method bodies 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 [14]) 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.

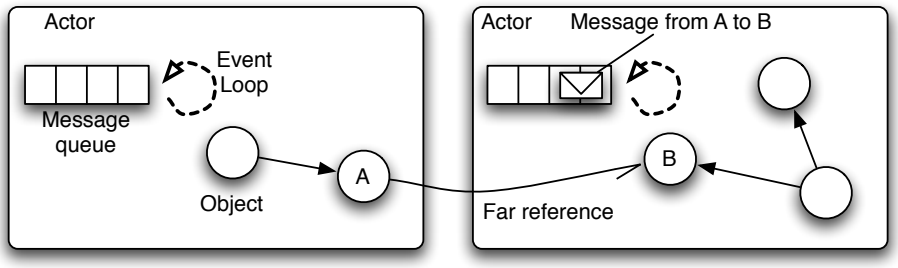


Figure 1: AmbientTalk actors as communicating event loops.

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 the need for 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 subsequent 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 sending 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");
} catch: { |e| ... }
```

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" [36] 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 fresh corresponding `resolver` object. The creator can then share a reference to the `future` object, 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 cause any callbacks registered on the future by means of the `when:becomes:catch:` function to be called in later event loop turns.

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 their fields.

No deadlocks If all message processing turns are finite, messages will never get stuck in actors' message queues. Processing a message in finite time is helped by the fact that an actor can never block and wait for a message in the middle of a turn. It can only receive messages between turns, and has no mechanism to selectively block certain messages from being processed. In addition, AmbientTalk futures are fully non-blocking. Contrary to most future abstractions in other languages, they do not support a blocking `get()` method to await the future's value synchronously. One can only await a future's value asynchronously, via a callback registered using the `when:becomes:catch:` function.

We should note that even in the absence of low-level data races and deadlocks, AmbientTalk programs can still exhibit higher-level safety and liveness issues. For example, while low-level data races are prevented, race conditions at the level of messages are still possible. For example, in the bank account example, if an object sends a `withdraw()` followed by a `deposit()` message, a third message sent by a different actor may interleave and be processed in between those two messages. The programmer is thus still responsible for synchronizing updates that are executed over *multiple* turns.

With respect to liveness, while messages do not get stuck in actors' message queues indefinitely, other forms of lost progress bugs are still possible. In particular, messages or notifications may still get stuck indefinitely in the message queues of futures. For example, one actor may register a callback on a future using `when:becomes:catch:`, but if this future is never resolved, the callback will never be executed. There is also a known issue with futures known as *data lock*, which occurs when a future gets resolved with itself (either directly or indirectly, via a cycle of futures depending on each other) [14]. However, contrary to deadlocks in a multithreaded program, these lost progress bugs are mostly deterministic and are thus easier to reproduce and debug.

We revisit these safety and liveness properties in more detail in Section 8.4.

6. Distributed AmbientTalk

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

Actors are AmbientTalk's unit of concurrent and distributed execution. A single AmbientTalk virtual machine (VM) may host multiple actors. A network may in turn connect multiple AmbientTalk VMs. 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). Whether two actors are running inside the same VM or not is not visible at the language level. The programmer should always consider two actors to be distributed, and prepare for possible failures, even if the actors may reside on the same VM.

Failure model. The VM is AmbientTalk's unit of failure: if a device running an AmbientTalk VM crashes, all the actors on that device crash. AmbientTalk provides no built-in abstractions to recover from device failures (e.g. actors have no persistent state). The network that connects multiple AmbientTalk VMs may also be subject to failures. At the implementation level, we assume a fully asynchronous network model where there is no upper bound on message delivery and individual network messages may be lost, duplicated and reordered. However, as we will explain in the following sections, at the programming language level, AmbientTalk makes the network reliable. Messages sent between remote actors will eventually be delivered, without duplicates and in FIFO order, as long as the network eventually heals and the VMs hosting the actors do not crash. Our message delivery protocol assumes cooperating VMs, we do not attempt to overcome Byzantine failures [37].

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[4]. 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 [21]. A subscription is made by registering a callback

[4]The current implementation uses a custom peer-to-peer service discovery protocol based on UDP and IP multicasting and is designed primarily for ad hoc WiFi networks. An AmbientTalk version that uses Bluetooth is also available.

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:

```
def sub := 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. The `whenever:discovered:` function returns an object `sub` 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 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 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 [38]. Objects can be exported with a *lease* such that any far reference that points to it provides access for only a limited period of time (the lease period). For example, the `MatchMaker` application can export the `myEndPoint` as a *leased object* instead of directly exporting the object itself:

```
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. When the proxy is passed to a client in another actor, a leased far reference (also called a *leased reference*) is created to the leased object which remains valid for at most 2 minutes. The client accesses the `myEndPoint` object *transparently* via a leased reference until the lease time elapses. 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 leased object.

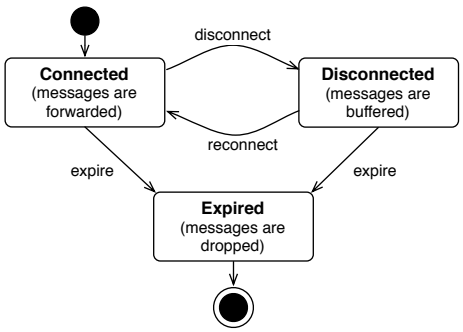


Figure 2: State diagram of a (leased) far reference.

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

6.3. Asynchronous Message Passing and Partial Failures

AmbientTalk also integrates leasing into asynchronous message passing to allow developers to specify an upper bound on how long to wait for the return value of a message. An asynchronous message may be annotated with the `Due` type tag. Like `TwoWay`, this indicates that the message send will return a future. The `Due` type tag expects a duration, which indicates an upper bound on how long this future will remain unresolved. We can use the `Due` type tag in the `exchange` message send to denote the time the application is willing to wait for the exchange of profiles as follows:

```
whenever: MatchMaker discovered: { |remoteEndPoint|
  def leasedFuture := remoteEndPoint<-exchange(myProfile)@Due(2.minutes);
  when: leasedFuture becomes: { |profile|
    // display profile info from remote peer
  } catch: TimeoutException using: { |e|
    // profile transfer failed
  }
};
```

The `Due` tag makes the `exchange` message send immediately return a future (stored in the variable `leasedFuture`). This future is passed *by leased reference* to the `remoteEndPoint` object initialized with a lease of 2 minutes. If the `leasedFuture` is resolved before its lease time has elapsed, the `becomes:` block triggers as usual. Otherwise, when the lease time elapses, the future is automatically ruined with a `TimeoutException` exception, and the `catch:` block is triggered. If the associated message is still buffered locally, it is removed from the buffer and discarded.

7. AmbientTalk at Work: Developing a Mobile Peer-to-peer Application

So far, we have shown how the language can be used to develop a small mobile peer-to-peer application. Over the past few years, we have employed AmbientTalk to build a variety of mobile peer-to-peer applications ranging from a drawing application[5] to various multiplayer games including an "urban game" called *Flikken* [39].

In this section, we describe a mobile music player application which we employ to compare AmbientTalk with Java RMI [6]. Java RMI can still be considered a state-of-the-art middleware for distributed object-oriented computing, forming the basis for mobile computing middelware such as Jini (cf. section 9).

[5] A demo of the drawing application called weScribble is available at `http://goo.gl/GUbJsD`

7.1. The Mobile Music Player Application

Consider a music player running on mobile devices. The music player contains a library of songs. When two people using the music player enter one another's personal area network, the music players set up an ad hoc network and exchange their music library's song index (not necessarily the songs themselves). After the exchange, the music players calculate the percentage of songs both users have in common. If this percentage exceeds a certain threshold, the music players can e.g. inform the user that someone with a similar taste in music is nearby.

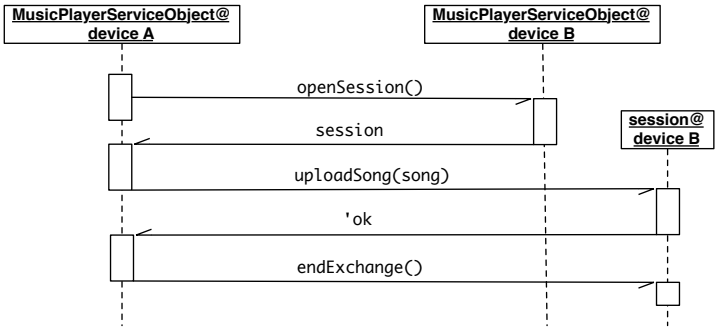


Figure 3: The music library exchange protocol

Figure 3 gives a graphical overview of the music library exchange protocol modeled in an asynchronous distributed object-oriented system. The figure depicts the protocol from the point of view of device `A`. This protocol is executed simultaneously on both devices. Once both devices discover each other, the music player running on `A` sends the `openSession` message to the remote peer `B` to start a session to exchange its library index. In response to it, the remote peer `B` returns a new `session` object which implements methods that allow the remote music player to send song information (`uploadSong`) and to signal the end of the library exchange (`endExchange`).

When implementing the music library exchange protocol, it is important to take the effects of volatile connections into account. First, the application should remain responsive in the face of intermittent failures, e.g., the application may want to inform the end-user that the transmission of songs is temporarily suspended. In addition, the application must also cope with permanent failures. Otherwise, if a peer disconnects in the middle of the library exchange, the session may never terminate, and the application will consume unnecessary resources, e.g., a partially uploaded library.

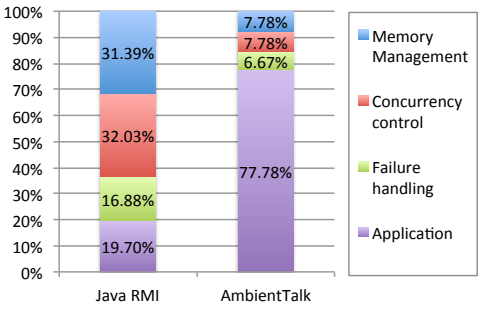
7.2. Comparison against Java RMI

We now compare AmbientTalk with Java RMI based on implementing the same music player application in both platforms and subsequently, we evaluate both implementations *quantitively* based on an analysis of their lines of code. We refer the reader to [40] for details regarding both the AmbientTalk and Java RMI implementations.

The AmbientTalk implementation (90 LoC) is significantly shorter than the corresponding Java RMI implementation (462 LoC). This difference can be partly attributed to the different syntactic nature of AmbientTalk and Java, so by itself these numbers do not mean much. However, what we can compare is the relative amount of code spent on non-functional concerns related to concurrency and distribution.

	Java RMI	AmbientTalk
Memory management code	145	7
Concurrency control code	148	7
Failure handling code	78	6
Application-level code	91	70
Total lines of code	**462**	**90**

(a) Absolute lines of code



(b) Relative percentages

Figure 4: (a) Absolute LoC and (b) relative % of four concerns in the music player: memory management, concurrency control, failure handling, application-specific code.

Table 4a lists a breakdown of the total number of lines of code according to four different concerns: 1) memory management: includes the code to setup leases and reclaim resources upon lease expiration, 2) concurrency control: includes support for asynchronous communication and buffering during transient disconnections, 3) failure handling: managing timeouts and 4) application-specific code.

Figure 4b shows the relative percentage of these lines of code compared to the total application size. Note the relatively small percentage of AmbientTalk LoC spent on the non-functional concerns. The main causes for the relative blow-up in non-functional code in Java RMI is that it does not feature buffered, asynchronous communication, and its support for leasing is very tightly coupled to the distributed garbage collection module. We should note that we deliberately left out any comparison regarding service discovery, as Java RMI has no support for this.

The only conclusion that we want to draw from this comparison is that AmbientTalk seems to succeed at reducing the impact of non-functional concerns that arise when developing fault-tolerant mobile P2P applications.

8. An Operational Semantics for Featherweight AmbientTalk

Our exposition of AmbientTalk so far was largely informal. The rest of the paper aims to provide a precise description of AmbientTalk's concurrency and distribution model. To this end, we present a small step operational semantics of a subset of AmbientTalk named *Featherweight AmbientTalk* or AT^f for short. Our semantics is based primarily on that of JCobox [41], 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^f subset contains actors, objects, isolates (pass-by-copy objects), blocks (functions), non-blocking futures and asynchronous message sending. In Section 8.5 we extend AT^f with service discovery, enabling objects in different actors to discover one another. In Section 8.6, we introduce the notion of disconnected actors and fault-tolerant messaging between actors. AT^f does not model AmbientTalk's object inheritance using prototype-based delegation, its support for trait-based composition (which is formalised elsewhere [29]), reflection, exceptions, leasing and JVM interoperability.

A machine-executable implementation of the AT^f semantics in PLT Redex [42] is available online [6].

8.1. Syntax

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

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

Figure 5: Abstract syntax of AT^f.

AT^f features both functional and imperative object-oriented elements. The functional elements descend directly from the λ-calculus (using an eager evaluation strategy). Anonymous functions are denoted by $\lambda x.e$ and correspond to AmbientTalk blocks. Variable lookup in AT^f is lexically scoped. Local variables can be introduced via let $x = e$ in e.

The imperative object-oriented elements stem from object-based (i.e. classless) calculi [43]. AT^f 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.

Futures can be created using the expression let $x_f, x_r =$ future in e. This expression brings *two* variables in scope within the subexpression e: x_f will be bound to a fresh future value and x_r will be bound to a corresponding resolver object. This AT^f expression corresponds to the built-in `makeFuture()` function from AmbientTalk (cf. Section 5.4).

The resolver object can be used to assign a value to its corresponding future, using the expression resolve x_r e. This resolves the future corresponding to x_r with the value of e. The resolve expression from AT^f corresponds to invoking the built-in `resolver.resolve(v)` method in AmbientTalk (cf. Section 5.4).

The value of a future x_f can be awaited using the expression 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.

[6] `http://soft.vub.ac.be/~cfscholl/index.php?page=at_semantics`

This AT^f expression corresponds to the `when:becomes:catch:` function in AmbientTalk (cf. Section 5.4).

AT^f 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 (cf. Section 5.4).

8.1.1. Syntactic Sugar

A number of AT^f expressions can be defined in terms of a desugaring (local transformation), as shown in Figure 6.

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

Figure 6: AT^f syntactic sugar.

It is well-known that functions can be expressed in terms of objects and vice-versa. AT^f 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 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^f program cannot fabricate these messages other than by using the "when" and "resolve" expressions.

The expression $\mathrm{when}(e \rightarrow x)\{e'\}$ 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 $\mathrm{resolve}\, e\, e'$ 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 its x argument bound to the value of e'. If e' reduces to a future value, the callback function will be called later, with its x argument bound to the resolved value of the future. Thus, this definition ensures that futures can only be truly resolved with non-future values.

8.2. Semantic Entities

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

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

Figure 7: Semantic entities of AT^f.

AT^f semantic entities are shown in Figure 7. 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^f 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 $t = \mathrm{O}$ denoting an object and $t = \mathrm{I}$ 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^f 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.

8.3. Reduction Rules

8.3.1. Evaluation Contexts

We use evaluation contexts [44] 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.

8.3.2. Notation

Actor heaps O are sets of objects, resolvers and futures. To lookup and extract values from a set O, we use the notation $O = O' \uplus \{o\}$. 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 $\emptyset$.

8.3.3. Evaluation Rules

Our semantics is defined in terms of a relation on configurations, $K \rightarrow K'$. 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.

An AT^f program is an expression e that is reduced in an initial configuration containing a single "main" actor $K_{init} = \{\mathcal{A}\langle \iota_a, \emptyset, \emptyset, [\text{null}/\text{self}]e\rangle\}$. The actor's heap and queue are initially empty, and the `self`-pseudovariable is bound to `null` at top-level.

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 8.6. 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 8: 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 9:

- LET: a "let"-expression simply substitutes the value of x for v in e according to the substitution rules outlined in Figure 8.
- 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

(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)

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

(MAKE-FUTURE)

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

(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)

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

(PROCESS-MSG-TO-FUTURE)

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

(RESOLVE)

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

Figure 9: Actor-local reduction rules.

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 10) 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 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 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:

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

Figure 10: Auxiliary functions used in the reduction rules.

  - 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, which must be a non-future value (i.e. $v \neq \iota_{a'}.\iota_{f'}$). At the same time, the future's queue Q' is emptied. The messages previously stored in 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 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 to the future's resolved value v.

Actor-global reductions. We summarize the actor-global reduction rules in Figure 11:

(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 11: Actor-global reduction rules.

- 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 10). 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 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.

8.3.4. Parameter-passing rules

The auxiliary function $pass(\iota_a, O, \overline{v}, \iota'_a)$ (see Figure 12) 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^f 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 stop 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 ($v = \iota_a.\iota_o, \mathcal{O}\langle \iota_o, \text{O}, F, M\rangle \in O$).

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.

8.4. Safety and Liveness Properties of AT^f

In Section 5.5 we argued that AmbientTalk's event loop concurrency model avoids low-level data races and deadlocks by design. Having introduced an operational semantics for AT^f, we can state these properties more formally.

8.4.1. Data Race Freedom

A data race would occur when two or more actors would simultaneously try to access the field of an object, and at least one of the field accesses is a field update. In the operational semantics, there are no true simultaneous accesses as the "concurrent" actors are reduced one step at a time. However, a simultaneous field access would occur if there are at least two actors in a configuration that would have a field access as the next expression to evaluate, and both actors could proceed. Formally:

Definition 1 (Field Access)**.** *A configuration K has a field access on a reference r in actor $a \in K$, denoted* $\text{fieldaccess}(K, a, r)$ *iff $a = \mathcal{A}\langle \iota_a, O, Q, e_\Box[e]\rangle$ with $e \equiv r.f$ or $e \equiv r.f := v$, and $K \rightarrow_k K''$, with $K'' = K'''\cup a'$ and $a' = \mathcal{A}\langle \iota_{\iota_a}, O, Q, e_\Box[v]\rangle$.*

Definition 2 (Simultaneous Field Access)**.** *A configuration K has a simultaneous field access on a reference r iff $K = K'\cup a \cup a'$ and* $\text{fieldaccess}(K, a, r)$ *and* $\text{fieldaccess}(K, a', r)$.

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

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

Figure 12: Auxiliary functions defining parameter-passing.

Theorem 1 (Data Race Freedom)**.** *Let K be a configuration. K is free of data races as there can be no simultaneous field accesses in K.*

Proof. We show that if there is more than one actor in a configuration trying to access the same field of the same object, then at most one of these actors can make a step. Let r be a reference to the object that is being simultaneously accessed. Let $K' \subseteq K$ be the set of actors performing a field access of the form $e_\Box[r.f]$ or $e_\Box[r.f := v]$. Then at most one of these field accesses can be reduced to a value v, all other actors will be stuck.

The only rules that allow such field accesses to be further reduced are (FIELD-ACCESS) and (FIELD-UPDATE). These rules can only be triggered if r is of the form $\iota_a.\iota_o$. If r is any value other than a reference to an object, e.g. null or $\iota_a.\iota_f$, the actor will get stuck.

If $r = \iota_a.\iota_o$ both rules require that the actor executing the expression has a corresponding identifier ι_a. As actor identifiers are unique within a configuration (cf. rule (NEW-ACTOR)), there can be at most one actor with identifier ι_a. Thus there can be at most one actor where $e_\Box[r.f]$ or $e_\Box[r.f := v]$ can be reduced to a value. All other actors executing these expressions must have different identifiers, and thus the (FIELD-ACCESS) and (FIELD-UPDATE) rules are not applicable, causing these actors to get stuck, thus preventing a simultaneous field access. □

8.4.2. Deadlock Freedom

We show that, given two reasonable assumptions, any message sent from one actor to another must eventually be processed. That is, messages cannot get stuck indefinitely in the message queues of actors, i.e. message passing in AT^f is deadlock-free. This result, however, depends on two assumptions:

Totality We assume that the individual message processing turns of all actors are *total*, i.e. that processing a message does not diverge or get stuck, but always eventually reduces to a value. This assumption certainly does not hold for all AT^f programs. It only holds for correct programs that do not provoke runtime errors (which would lead to stuck states) and that do not diverge (i.e. go into an infinite loop).

Axiom 1 (Totality)**.** *A total actor configuration K is one where each actor $a \in K$ that is evaluating a compound expression e will eventually reduce this expression to a value v. That is, $\forall a \in K, a = \mathcal{A}\langle \iota_a, O, Q, e\rangle, e \neq v \; \exists K' : K \uplus a \rightarrow_k^* K' \cup \mathcal{A}\langle \iota_a, O', Q', v\rangle$*

Here, $\rightarrow_k^*$ is the reflexive transitive closure of $\rightarrow_k$.

Fairness Actor configurations K can be reduced in a non-deterministic manner when multiple reduction rules are applicable, as we do not specify a particular order in which the rules apply. However, we will assume a *fair* scheduling strategy among actors (i.e. every actor that *can* make progress eventually *does* make progress). An unfair scheduler could prevent an actor from making progress by always choosing reduction rules that advance only other actors. Fairness can be enforced by the rewrite system (for AT^f) or the VM (for AmbientTalk).

Axiom 2 (Fairness)**.** *We require AT^f computations to be fair. A computation is fair if each reduction rule $K \rightarrow_k K'$ that is enabled (i.e. is applicable) is either eventually chosen or becomes permanently disabled (i.e. inapplicable).*

Our definition of fairness is based on Agha et al.'s foundational work [45]. We refer to their work for a more formal and detailed treatment of fairness.

Given these assumptions, it can be guaranteed that messages never get stuck in the message queues of actors, i.e. all messages are eventually processed. Our proof depends on the following two lemmas:

Lemma 1 (Message sending is non-blocking)**.** *Sending an asynchronous message never blocks the sending actor:* $\forall K \cup a, a = \mathcal{A}\langle \iota_a, O, Q, e_\Box[\iota_{a'}.\iota_{o'} \leftarrow m(\overline{v})]\rangle \; \exists K' : K \cup a \rightarrow_k K' \cup \mathcal{A}\langle \iota_a, O, Q', e_\Box[\text{null}]\rangle$.

Proof. If $\iota_{a'} = \iota_a$ then the message send is local, and the (LOCAL-ASYNCHRONOUS-SEND) rule in combination with the (CONGRUENCE) rule is directly applicable.

Otherwise, $\iota_{a'} \neq \iota_a$ such that the only rule that is applicable is (FAR-ASYNCHRONOUS-SEND) This rule is applicable regardless of what particular expression e' the recipient actor $\iota_{a'}$ is executing. The recipient actor does not need to be in any particular state to accept new incoming messages. □

Lemma 2 (Message receipt is non-selective)**.** *If an actor is ready to process a message and its message queue is non-empty, it* must *always process the next message. It cannot skip or delay the processing of certain messages. That is* $\forall K \cup a, a = \mathcal{A}\langle \iota_a, O, Q \cdot \mathcal{M}\langle v, m, \overline{v}\rangle, v'\rangle \; \exists K' : K \cup a \rightarrow_k K' \cup \mathcal{A}\langle \iota_a, O', Q, e\rangle$.

Proof. By case analysis on the form of v, the receiver of the message. A value v can either be a reference r or the values null or ϵ. The latter two can never be a valid receiver value: the only rules that ever enqueue messages in an actor's queue are (LOCAL-ASYNCHRONOUS-SEND), where the value must be a local reference, and (FAR-ASYNCHRONOUS-SEND), where the value must be a far reference. It follows that v must be a reference, more specifically, a reference $\iota_a.\iota_o$ denoting an object, future or resolver owned by a.

If v is a reference to an object or resolver, i.e. $v = \iota_a.\iota_o$ with $\iota_o \notin \mathbf{FutureId}$, then the rule (PROCESS-MESSAGE) is applicable. Apart from requiring that the actor's expression has been fully reduced to a value v and that the message is first in the queue, there are no particular restrictions on the contents of the message.

If v is a reference to a future, i.e. $v = \iota_a.\iota_f$, then rule (PROCESS-MSG-TO-FUTURE) is applicable, regardless of whether the future is resolved or unresolved. □

Theorem 2 (Eventual Message Processing)**.** *All messages sent between actors are eventually processed. That is, messages cannot get stuck indefinitely in actors' message queues.*

Proof. By Lemma 1, when an actor a_1 is ready to send a message m it can always proceed independent of the state of the recipient actor a_2. By Axiom 2, the rule (FAR-ASYNCHRONOUS-SEND) will eventually be applied. The message is now in the incoming message queue Q of a_2.

We now show that the message m is eventually processed by a_2. First, assume $Q = Q' \cdot m$ (i.e. m is the first message in the queue). If a_2 is idle (i.e. it has fully reduced its expression to a value v), Lemma 2 is applicable and a_2 can eventually process m (Axiom 2). Otherwise, a_2 is still processing an earlier message. By Axiom 1, message processing is total such that the actor eventually becomes idle, reducing the problem to the previous case.

If Q contains more than one message, i.e. $Q = Q' \cdot m'$, the above reasoning similarly applies and m' will eventually be processed, shortening a_2's queue to Q'. As messages are processed in strict FIFO order, m must eventually become the first message in the queue and will eventually be processed. □

8.5. Service Discovery

In Figure 13, we extend AT^f with primitives for service discovery, allowing objects in different actors to discover one another as described in Section 6.1.

Semantic Entities

$$\begin{array}{rcl} a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, O, Q, X, I, e\rangle \\ X \subseteq \textbf{Exports} & ::= & (O, v, \tau) \\ I \subseteq \textbf{Imports} & ::= & (r, \tau) \\ v \in \textbf{Value} & ::= & \ldots \mid \tau \\ \tau \in \textbf{Type} & & \end{array}$$

Syntax

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

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

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

Figure 13: Extensions for service discovery.

AT^f actors are extended with a set of exported objects X 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^f 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 14 lists the additional reduction rules for service discovery:

(PUBLISH)

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

(SUBSCRIBE)

$$\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, X, I, e_\Box[\text{discover } \tau\ \iota_a.\iota_o)]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, X, I \cup (\iota_a.\iota_o, \tau), e_\Box[\text{null}]\rangle\end{array}$$

(MATCH)

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

Figure 14: 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 actor's set of exported objects X 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.

Note that when exporting an isolate object, the isolate will be copied at least twice: once when initially exported, and once again whenever it is matched against a subscription. The initial copy upon export is necessary such that the exported object becomes self-contained, and can no longer be modified by the exporting actor. The (MATCH) rule can be triggered while the exporting actor is in the middle of evaluating an expression e'. If we were to make the initial copy of the exported object in the (MATCH) rule, a data race could be possible where the actor $\iota_{a'}$ was in the middle of updating the isolate's fields.

8.6. Robust time-decoupled message sends

So far, we have assumed that AT^f 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 15, we extend AT^f 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 15: 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* [21], 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 16 lists the additional reduction rules for time-decoupled message transmission:

(FAR-ASYNCHRONOUS-SEND)
This rule is replaced by (CREATE-MESSAGE) and (TRANSMIT-MESSAGE).

(CREATE-MESSAGE)

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

(TRANSMIT-MESSAGE)

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

(MOBILITY)

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

Figure 16: Reduction rules for time-decoupled message sends.

- 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.

8.7. AT^f *compared to AmbientTalk*

The extensions to AT^f for service discovery and robust time-decoupled message sends do not break the results established in Section 8.4 pertaining to data race and deadlock freedom, with one exception: to uphold deadlock-freedom, time-decoupled message sends introduce the additional assumption that the sender and receiver actor must eventually join the same network. Otherwise, the message cannot be delivered and will remain stuck in the sender's outgoing message queue.

While AT^f only models the essential core language of AmbientTalk, the race and deadlock freedom results ought to be transferrable to the full language, as we have faithfully modelled all communication primitives present in the full language. None of AmbientTalk's libraries or built-in functions introduce additional ways for two actors to directly share state or to communicate other than by asynchronous message passing.

9. 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 three broad categories: languages and systems also directed at developing software for mobile networks, related work in actor-based languages in general and related work on the formalization.

Mobile networks. AmbientTalk tackles the issues of mobile networks by building on object-oriented abstractions such as object references and message-passing. Jini [7] is the closest object-oriented approach tailored to peer-to-peer networks. Its main goal is to allow clients and services to discover and set up an ad hoc network in a flexible, easy manner with minimal administrative infrastructure. To this end, it introduces the notion of lookup services. However, once services have been introduced, Jini relies on the synchronous communication model of Java RMI. Jini also employs the concept of leasing to allow devices leave the network gracefully without affecting the rest of the system.

Others have tackled the same issues by building on different communication paradigms. For example, LIME [46] and TOTA [47] are mobile computing middleware based on tuple spaces [48]. 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 [21]. 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) [49] and STEAM [50], 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* [2]. 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 [14], 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 [51], 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 [18]. In Argus, promises additionally supported pipelined message sends and exceptions [20]. Most future abstractions support synchronisation by suspending a thread that accesses an unresolved future. This style of synchronization is called *wait-by-necessity* [19]. The E language pioneered the `when`-expression to await the value of a promise in a non-blocking way [14]. In other actor systems, the same goal is often accomplished by passing explicit callbacks or "continuation" actors as arguments to a message. The Salsa language, for instance, uses "token-passing continuations" to express the follow-up processing of an asynchronous message [32].

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 [52].

The view of AmbientTalk actors as containers of regular objects is based on E's similar notion of actors as *vats* [14]. In JCoBox [41], actors are similarly represented as *coboxes*. JCoBox additionally supports cooperative multitasking (coroutines) within a cobox. AmbientTalk's isolates (pass-by-copy objects) are similar to E "pass-by-construction" objects and JCoBox transfer objects.

Process calculi. The π-calculus [53] models processes that communicate over channels. Channels are *mobile* in the sense that a channel can be passed as argument over another channel. This leads to a dynamically reconfigurable network of processes, similar to what can be achieved in the actor model by passing the mail address of one actor as an argument in a message to another actor. Contrary to the actor model, communication over channels is primitively synchronous rather than asynchronous (even though asynchronous communication can be modelled in the π-calculus, just as synchronous communication can be modelled in the actor model).

The ambient calculus [54] is a calculus designed to model mobility of processes and devices. AmbientTalk does not focus on mobile actors (processes) but only on the mobile devices that host them. Thus, the networks introduced in Section 8.6 to some extent play the role of mobile ambients in the ambient calculus, implicitly describing which actors are co-located and thus able to communicate. Unlike mobile ambients, our networks cannot be hierarchically tree-structured.

Nomadic Pict [55] is a programming model and a language designed for distributed and mobile computation. Like the ambient calculus, it mainly focuses on mobile *computation*, where mobile agents can be migrated across locations. Nomadic Pict distinguishes between location-dependent communication primitives that require knowledge about the site of an agent and location-independent primitives that enable communication with an agent without requiring knowledge of its current site. AmbientTalk actors, unlike agents, are immobile, even though iso-

late objects can be copied between actors, introducing a form of code and data mobility without identity (the copied isolate has its own identity).

The calculus of asynchronous sequential processes (ASP) [56] models parallel activities (similar to actors) that interact only through asynchronous method calls. Contrary to our communicating event loops model, which may host multiple remotely addressable objects inside a single actor, in ASP, each activity contains a single remotely addressable object (an "active object"). All other objects are called "passive" objects, and are not remotely addressable (like AmbientTalk's isolates, they are passed by deep copy between activities). ASP also introduces futures, but synchronization on a future is implicit and blocking, in the sense that the activity is suspended when it needs to reduce an expression that needs the value of the future (this is called *wait-by-necessity*). By contrast, awaiting the value of a future is explicit and non-blocking in AmbientTalk, by registering a callback function using `when:becomes:catch:`. Just like AT^f models the essential concurrency features of AmbientTalk, so ASP models the essential features of ProActive [34], a Java middleware for distributed object-oriented programming.

Formalization. The notion of formalizing an actor system as transitions on actor configurations dates back to the original formalization of actors by Agha et al. [45]. However, the operational semantics of AT^f is primarily based on that of JCoBox [41]. Whereas Agha et al. use a functional base language, JCoBox and AT^f use an imperative object-oriented base language. JCoBox itself is based on Featherweight Java [57]. We adapted the semantics to instead use a dynamically typed, classless base language, and to reflect AmbientTalk's communicating event loops model with non-blocking futures. The proof for data race freedom in 8.4.1 is analogous to the proof for the same property in JCoBox [58]. The by-copy parameter-passing rules for isolates are similar to the parameter-passing rules for passive objects in ASP [56].

10. Conclusion

Developing mobile peer-to-peer applications is challenging due to 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.

We presented AmbientTalk, an actor-based language designed for developing mobile P2P applications that remain resilient to network failures by default. AmbientTalk extends the actor model with support for service discovery and fault-tolerant message sending. By comparison with standard OO middleware such as Java RMI, AmbientTalk programs require less lines of code spent on non-functional requirements.

We presented a small-step operational semantics for Featherweight AmbientTalk, a subset modelling the core features of the language. We established data race freedom and deadlock freedom as the key concurrency properties of the communicating event loops model, on which AmbientTalk is based.

References

[1] C. Mascolo, L.Capra, W. Emmerich, Mobile Computing Middleware, in: Advanced lectures on networking, Springer-Verlag, 2002, pp. 20–58.

[2] G. Agha, Actors: a Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[3] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter, Ambient-oriented Programming in AmbientTalk, in: D. Thomas (Ed.), Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP), Vol. 4067 of Lecture Notes in Computer Science, Springer, 2006, pp. 230–254. doi:http://dx.doi.org/10.1007/11785477_16.

[4] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter, Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks, in: Inter. Conf. of the Chilean Computer Science Society (SCCC), IEEE Computer Society, 2007, pp. 3–12.

[5] E. G. Boix, T. V. Cutsem, J. Vallejos, W. D. Meuter, T. D'Hondt, A leasing model to deal with partial failures in mobile ad hoc networks., in: M. Oriol, B. Meyer (Eds.), TOOLS (47), Vol. 33 of Lecture Notes in Business Information Processing, Springer, 2009, pp. 231–251.

[6] Sun Microsystems, Java RMI specification, `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html` (1998).

[7] J. Waldo, Constructing ad hoc networks, in: IEEE International Symposium on Network Computing and Applications (NCA'01), 2001, p. 9.

[8] J. Armstrong, R. Virding, C. Wikström, M. Williams, Concurrent Programming in Erlang, 2nd Edition, Prentice-Hall, 1996.
URL `citeseer.comp.nus.edu.sg/393979.html`

[9] E. Jul, H. Levy, N. Hutchinson, A. Black, Fine-grained mobility in the Emerald system, ACM Transactions on Computer Systems 6 (1) (1988) 109–133.
URL `citeseer.ist.psu.edu/jul88finegrained.html`

[10] B. Liskov, Distributed programming in Argus, Communications Of The ACM 31 (3) (1988) 300–312. doi:http://doi.acm.org/10.1145/42392.42399.

[11] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Conference proceedings on Object-oriented Programming Systems, Languages and Applications, ACM Press, 1987, pp. 227–242. doi:http://doi.acm.org/10.1145/38765.38828.

[12] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison-Wesley Longman Publishing Co., Inc., 1989.

[13] G. J. Sussman, G. L. S. Jr., Scheme: An interpreter for extended lambda calculus, in: MEMO 349, MIT AI LAB, 1975.

[14] M. Miller, E. D. Tribble, J. Shapiro, Concurrency among strangers: Programming in E as plan coordination, in: Symposium on Trustworthy Global Computing, Vol. 3705 of LNCS, Springer, 2005, pp. 195–229.

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

[16] E. A. Lee, The problem with threads, Computer 39 (5) (2006) 33–42. doi:10.1109/MC.2006.180.
URL `http://dx.doi.org/10.1109/MC.2006.180`

[17] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, M. F. Kaashoek, Rover: a toolkit for mobile information access, in: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), Colorado, 1995, pp. 156–171.

[18] A. Yonezawa, J.-P. Briot, E. Shibayama, Object-oriented concurrent programming in ABCL/1, in: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press, 1986, pp. 258–268. doi:http://doi.acm.org/10.1145/28697.28722.

[19] D. Caromel, Towards a method of object-oriented concurrent programming, Communications of the ACM 36 (9) (1993) 90–102.
URL `citeseer.ist.psu.edu/300829.html`

[20] B. Liskov, 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, ACM Press, 1988, pp. 260–267. doi:http://doi.acm.org/10.1145/53990.54016.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, A. Kermarrec, The many faces of publish/subscribe, ACM Computing Survey 35 (2) (2003) 114–131. doi:http://doi.acm.org/10.1145/857076.857078.

[22] A. Kaminsky, H.-P. 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), ACM Press, 2002, pp. 72–73. doi:http://doi.acm.org/10.1145/985072.985109.

[23] G. Bracha, D. 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, 2004, pp. 331–343.

[24] T. Watanabe, A. Yonezawa, Reflection in an object-oriented concurrent language, in: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press, 1988, pp. 306–315. doi:http://doi.acm.org/10.1145/62083.62111.

[25] J. McAffer, Meta-level programming with coda, in: ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming, Springer-Verlag, London, UK, 1995, pp. 190–214.

[26] T. Van Cutsem, S. Mostinckx, W. De Meuter, Linguistic symbiosis between event loop actors and threads, Computer Languages Systems & Structures 35 (1) (2009) 80–98. doi:10.1016/j.cl.2008.06.005.

[27] S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Gonzalez Boix, E. Tanter, W. De Meuter, Mirror-based reflection

in AmbientTalk, Softw. Pract. Exper. 39 (7) (2009) 661–699. doi:http://dx.doi.org/10.1002/spe.v39:7.
[28] N. Schärli, S. Ducasse, O. Nierstrasz, A. P. Black, Traits: Composable units of behaviour, in: L. Cardelli (Ed.), ECOOP, Vol. 2743 of Lecture Notes in Computer Science, Springer, 2003, pp. 248–274.
[29] T. Van Cutsem, A. Bergel, S. Ducasse, W. 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, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 220–243. doi:http://dx.doi.org/10.1007/978-3-642-03013-0_11.
[30] H. Lieberman, Concurrent object-oriented programming in ACT 1, in: A. Yonezawa, M. Tokoro (Eds.), Object-Oriented Concurrent Programming, MIT Press, 1987, pp. 9–36.
[31] 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, ACM Press, New York, NY, USA, 1988, pp. 69–72. doi:http://doi.acm.org/10.1145/67386.67403.
[32] C. Varela, G. Agha, Programming dynamically reconfigurable open systems with SALSA, SIGPLAN Not. 36 (12) (2001) 20–34. doi:http://doi.acm.org/10.1145/583960.583964.
[33] S. Srinivasan, A. Mycroft, Kilim: Isolation-typed actors for java, in: ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 104–128. doi:http://dx.doi.org/10.1007/978-3-540-70592-5_6.
[34] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, Grid Computing: Software Environments and Tools, Springer-Verlag, 2006, Ch. Programming, Deploying, Composing, for the Grid.
URL http://www-sop.inria.fr/oasis/proactive/userfiles/file/papers/ProgrammingComposingDeploying.pdf
[35] P. Haller, M. Odersky, Actors that unify threads and events, in: A. Murphy, J. Vitek (Eds.), Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings, Vol. 4467 of Lecture Notes in Computer Science, Springer, 2007, pp. 171–190.
[36] J.-P. Briot, R. Guerraoui, K.-P. Lohr, Concurrency and distribution in object-oriented programming, ACM Computing Surveys 30 (3) (1998) 291–329.
[37] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401. doi:10.1145/357172.357176.
URL http://doi.acm.org/10.1145/357172.357176
[38] C. Gray, 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, ACM Press, New York, NY, USA, 1989, pp. 202–210. doi:http://doi.acm.org/10.1145/74850.74870.
[39] C. Scholliers, E. Gonzalez Boix, W. De Meuter, T. D'Hondt, Context-aware tuples for the ambient, On the Move to Meaningful Internet Systems, OTM 2010 (2010) 745–763.
[40] E. Gonzalez Boix, Handling partial failures in mobile ad hoc network applications: From programming language design to tool support, Ph.D. thesis, Vrije Universiteit Brussel, Faculty of Sciences, Software Languages Lab (October 2012).
[41] J. Schäfer, A. Poetzsch-Heffter, Jcobox: generalizing active objects to concurrent components, in: Proceedings of the 24th European conference on Object-oriented programming, ECOOP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 275–299.
URL http://dl.acm.org/citation.cfm?id=1883978.1883996
[42] M. Felleisen, R. B. Findler, M. Flatt, Semantics Engineering with PLT Redex, 1st Edition, The MIT Press, 2009.
[43] M. Abadi, L. Cardelli, A Theory of Objects, 1st Edition, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
[44] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theor. Comput. Sci. 103 (2) (1992) 235–271. doi:10.1016/0304-3975(92)90014-7.
[45] G. A. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, A foundation for actor computation, J. Funct. Program. 7 (1) (1997) 1–72. doi:10.1017/S095679689700261X.
URL http://dx.doi.org/10.1017/S095679689700261X
[46] A. Murphy, G. Picco, G.-C. Roman, LIME: A middleware for physical and logical mobility, in: Proceedings of the The 21st International Conference on Distributed Computing Systems, IEEE Computer Society, 2001, pp. 524–536.
[47] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications with the TOTA middleware, in: PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society, Washington, DC, USA, 2004, p. 263.
[48] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 80–112.
[49] P. Eugster, B. Garbinato, A. Holzer, Location-based publish/subscribe, Fourth IEEE International Symposium on Network Computing and Applications (2005) 279–282.
[50] R. Meier, V. Cahill, A. Nedos, S. Clarke, Proximity-based service discovery in mobile ad hoc networks, in: Distributed Applications and Interoperable Systems, Springer, 2005, pp. 115–129.

[51] C. J. Callsen, G. Agha, Open heterogeneous computing in ActorSpace, Journal of Parallel and Distributed Computing 21 (3) (1994) 289–300.
URL citeseer.ist.psu.edu/callsen94open.html

[52] V. A. Saraswat, Concurrent constraint programming, MIT Press, Cambridge, MA, USA, 1993.

[53] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, i, Inf. Comput. 100 (1) (1992) 1–40. doi:10.1016/0890-5401(92)90008-4.
URL http://dx.doi.org/10.1016/0890-5401(92)90008-4

[54] L. Cardelli, A. D. Gordon, Mobile ambients, in: Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98, Springer-Verlag, Berlin Germany, 1998.
URL citeseer.nj.nec.com/cardelli98mobile.html

[55] P. Sewell, P. T. Wojciechowski, A. Unyapoth, Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation, ACM Trans. Program. Lang. Syst. 32 (4) (2010) 12:1–12:63. doi:10.1145/1734206.1734209.
URL http://doi.acm.org/10.1145/1734206.1734209

[56] D. Caromel, L. Henrio, B. P. Serpette, Asynchronous and deterministic objects, in: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '04, ACM, New York, NY, USA, 2004, pp. 123–134. doi:10.1145/964001.964012.
URL http://doi.acm.org/10.1145/964001.964012

[57] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: a minimal core calculus for java and gj, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450. doi:10.1145/503502.503505.
URL http://doi.acm.org/10.1145/503502.503505

[58] J. Schäfer, A programming model and language for concurrent and distributed object-oriented systems, Ph.D. thesis, Technischen Universität Kaiserslautern (2010).