An Operational Semantics of Event Loop Concurrency in AmbientTalk

Tom Van Cutsem[*] Christophe Scholliers[†] Dries Harnie
Wolfgang De Meuter
Software Languages Lab - Vrije Universiteit Brussel
Technical Report VUB-SOFT-TR-12-04

Abstract

We present an operational semantics of a key subset of the AmbientTalk programming language, AT-LITE. The subset focuses on its support for asynchronous, event-driven programming. AmbientTalk is a concurrent and distributed language that implements the so-called communicating event loops model.

1 Introduction

AmbientTalk is an actor-based programming language, designed specifically for a new class of computer networks, so-called *mobile ad hoc networks* [9]. These are networks populated by mostly mobile devices that communicate peer-to-peer using wireless communication technology, such as WiFi or Bluetooth. Thanks to the emergence of smartphone platforms such as iOS and Android, such networks have become omnipresent, and in this light AmbientTalk can best be summarized as a scripting language for mobile peer-to-peer apps.

In the next section, we give a brief overview of AmbientTalk, paying special attention to its concurrent and distributed programming model, which is founded on actors. Subsequently, we present an operational semantics for a key subset of AmbientTalk.

Availability An open-source interpreter for AmbientTalk is available at `http://ambienttalk.googlecode.com`. The interpreter is written in Java and runs on any Java 1.4-compliant or later JVM. A version for Android-powered phones is available. An executable version of the operational semantics described in this paper in PLT Redex is available from `http://soft.vub.ac.be/~cfscholl/AT-Redex-Model.zip`.

[*]Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO).

[†]Christophe Scholliers is funded by a doctoral scholarship of the Institute for the Promotion through Science and Technology in Flanders (IWT-Vlaanderen).

2 Communicating Event Loops

In AmbientTalk, concurrency is spawned by actors: one AmbientTalk virtual machine may host multiple actors which execute concurrently. AmbientTalk's concurrency model is based on the communicating event loops model of the E programming language [6], which is itself an adaptation of the well-known actor model [1]. The E language combines actors and objects into a unified concurrency model. Unlike previous actor languages such as Act1 [5], ABCL [10] and Actalk [2], actors are not represented simply as "active objects", but rather as *vats* or containers, encapsulating a set of regular objects.

Thus, actors are not represented as active objects, but rather as a collection of plain objects that share a single event loop. That event loop has a single message queue, containing messages addressed to its objects. The event loop perpetually takes a 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. Processing a single asynchronous message to completion in this way is called a *turn*. Turns are the basic unit of "event interleaving" in AmbientTalk: incoming messages are processed fully before any other message gets to 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.

In summary, messages are processed serially and to completion to avoid low-level race conditions on the mutable state of the contained objects. In AmbientTalk, each object is said to be *owned* by exactly one actor. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor may communicate using standard, sequential message passing or using asynchronous message passing. AmbientTalk borrows from the E language the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). It is possible for objects owned by one actor to refer directly to individual objects owned by another actor. Such references that span different actors are named *far references* (the terminology stems from E [6]) and only allow asynchronous access to the referenced object. Any messages sent via a far reference to an object are enqueued in the message queue of the owner of the object and processed by the owner itself.

Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the event loop processes of the actors which perpetually take messages from their message queue and synchronously execute the corresponding methods on the actor's owned objects. An event loop process never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. For example, when `A` sends a message to `B`, the message is enqueued in the message queue of `B`'s actor which eventually processes it.

Asynchronous messages can be sent between objects owned by the same actor (via a local reference) or by different actors (via a far reference). AmbientTalk features special

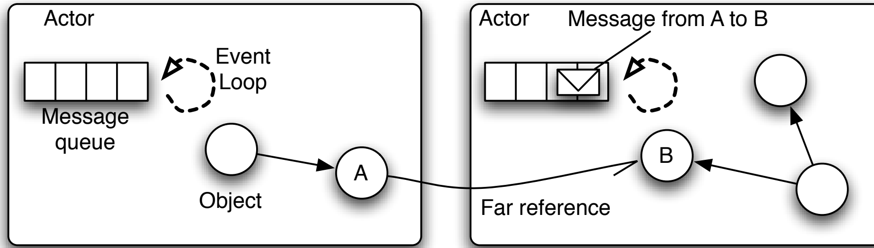

Figure 1: AmbientTalk actors as event loops

objects called *isolates* that are passed by copy when parameter-passed between actors.

AmbientTalk's concurrency model avoids low-level data races and deadlocks by design. It avoids low-level races because actors can only directly operate on their own objects. It avoids deadlocks because there is no blocking operation: both message sending and message reception are fully asynchronous. Unlike Erlang, AmbientTalk and E do not allow an actor to block and wait for messages to arrive in the middle of a turn.

2.1 Example: a Simple Echo Service

Listing 1 describes the code for a simple echo service, showcasing some of AmbientTalk's language features. The echo service object simply accepts any incoming string and returns it immediately to the sender.

Listing 1: Definition of a Simple Echo Service

```
// the singleton echo service object
def service := object: {
  def echo(str) {
    system.println("Received: "+str);
    str // return value is the same string
  }
}

// define the type of the service (on the server)
deftype EchoService;

// advertise the service in the network
def pub := export: service as: EchoService;
```

The code in listing 1 defines a singleton echo service object `service`, and subsequently advertises this service in the local ad hoc network. Service discovery operates by means of a simple publish/subscribe mechanism. Once an object is exported, the AmbientTalk VM takes care of broadcasting an advertisement to nearby listening VMs. Actors running on other AmbientTalk VMs can get notified of these advertisements via a subscription mechanism. On the client-side, a typical interaction with such an echo service is shown in listing 2.

Listing 2: A Simple Client Interaction

```
// define the type of the service (on the client)
deftype EchoService;

// discover the service
when: EchoService discovered: { |echoService|
  system.println("Discovered an echo service");

  // send an asynchronous message to the service
  def replyFuture := echoService<-echo("test message")@TwoWay;
  // the following call does not block the actor:
  when: replyFuture becomes: { |reply|
    // react to the incoming reply
    system.println("Reply: " + reply);
  }
}
```

The client actor interacts with the echo service in a fully asynchronous and event-driven way: it first registers a callback to be triggered when an echo service was discovered by the underlying AmbientTalk VM. When this event occurs, the client sends an asynchronous `echo` message, and awaits the reply.

The `@TwoWay` annotation signifies that the `echo` message should immediately return a *future* for the return value of the method. Futures are placeholders for asynchronously computed values. In AmbientTalk, as in E, an actor cannot suspend on a future until it is resolved. Instead, one can await the resolution of a future in a non-blocking way by posting a callback using the construct `when: future becomes: callback`. The callback is scheduled for execution in the actor, when the future is resolved. The single argument to the callback, `reply` in the above example, is then bound to the value with which the future is resolved. This design ensures that the actor remains responsive to other incoming messages while awaiting the value of a future.

3 Operational Semantics

Our exposition of the semantics of a subset of AmbientTalk, named AT-Lite, is based primarily on that of the Cobox model [8]. Coboxes feature a similar runtime model, but differ on important points such as the ability to execute multiple coroutines inside a single actor, and the ability to block (suspend) on a future. Our notion of futures as presented here is different from the notion of futures as presented in the Cobox model.

Our operational semantics models objects, isolates (pass-by-copy objects), actors as event loops, non-blocking futures, asynchronous message sending and the semantics of inter-actor parameter-passing. In Section 3.3 we extend AT-Lite with the necessary primitives for service discovery, enabling objects in different actors to discover one another as shown in the previous example.

3.1 Syntax

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

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

Figure 2: Semantic entities of AT-Lite.

Figure 3.1 lists the different semantic entities of AT-Lite. Caligraphic letters like $\mathcal{F}$ and $\mathcal{M}$ are used as "constructors" to distinguish the different semantic entities syntactically. Actors, futures, resolvers and objects each have a distinct address or identity, denoted ι_a, ι_f, ι_r and ι_o respectively.

Configurations are sets 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 process in the future, 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 so-called *isolate* objects, with $t = \textsc{o}$ denoting an object and $t = \mathrm{I}$ denoting an isolate. Isolates differ from regular objects in that they are parameter-passed by-copy rather than by-reference in inter-actor message sends, but otherwise behave the same.

AT-Lite supports *futures*, which are first-class objects that are placeholders for a value that is asynchronously awaited. 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.

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. Our notion of future-resolver pairs descends directly from E's promise-resolver pairs [6], which are themselves inspired by logic variables in concurrent constraint programming [7].

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 **FutureId** and **ResolverId** to be a subset of **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.

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

AT-LITE is also an imperative classless object-oriented language. It features `object` and `isolate` literal expressions to define fresh, anonymous objects. Objects consist of a set of fields and methods. Fields may be accessed and updated. Methods can be invoked either synchronously via $e.m(\overline{e})$ or asynchronously via $e \leftarrow m(\overline{e})$.

In the context of a method, the pseudovariable `this` refers to the enclosing object or literal. `this` 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 `this`-pseudovariable. That is, they must have $FV(e) \subseteq \{\text{this}\}$ for all field initialiser and method body expressions e. Isolates and actors are literally "isolated" from their surrounding lexical scope, making them self-contained.

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

AT-LITE supports two forms of asynchronous message passing. Expressions of the form $e \leftarrow m(\overline{e})$ denote one-way asynchronous message sends that do not return a useful value.

If a return value is expected, the expression $e \leftarrow_f m(\overline{e})$ denotes a two-way asynchronous message send that immediately returns a future for the result of invoking the method m.

Syntactic Sugar Functions are defined as objects with a single method called `apply`. The substitution $[x_{this}/\text{this}]e$ is necessary to ensure that within function bodies nested inside object methods, the `this`-pseudovariable remains bound to the original enclosing object, 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 such that the recipient actor can decode the argument list, knowing it will have to pass the result of the method invocation to x_r. The two-way message send itself evaluates to the future x_f corresponding to the passed resolver x_r.

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

The expression $\text{resolve}\, e\, e'$ is used to resolve a future with a value, where e must reduce to a resolver and e' to any value. If e' reduces to a non-future value, the listener function x_l will be called with x bound to the value of e'. If e' reduces to a future value, the listener function will be called later, with x bound to the resolved value of the future. Thus, this definition ensures that futures can only be truly resolved with non-future values.

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

Evaluation Contexts and Runtime Expressions We use evaluation contexts [4] to indicate what subexpressions of an expression should be fully reduced before the compound expression itself can be further reduced. $e_\square$ denotes an expression with a "hole". Each appearance of $e_\square$ indicates a subexpression with a possible hole. The intent is for the hole to identify the next subexpression to reduce in a compound expression.

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

Syntax

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

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

Syntactic Sugar

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

Evaluation Contexts and Runtime Syntax

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

$$
e \quad ::= \quad \ldots \mid r
$$

Substitution Rules

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

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

Figure 3: Substitution rules: x denotes a variable name or the pseudovariable this.

3.2 Reduction Rules

Notation Actor heaps O are sets of objects, resolvers and futures. To lookup and extract values from a set O, we use the notation $O = O' \cup \{o\}$. This splits the set O into a singleton set containing the desired object o and the disjoint set $O' = O \setminus \{o\}$. The notation $Q = Q' \cdot \mathrm{m}$ deconstructs a sequence Q into a subsequence Q' and the last element m. In AT-Lite, queues are sequences of messages and 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 using $\emptyset$. 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.

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 a next message in the middle of a reduction.

If no actor-local reduction rule is applicable to further reduce a reducible expression, this signifies an error in the program. 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.

We now summarize the actor-local reduction rules in Figure 3.2:

- LET: a "let"-expression simply substitutes the value of x for v in e according to the substitution rules outlined in Figure 3.1.

- 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. The effect of evaluating an object or literal expression is the addition of a new object to the actor's heap. The fields of the new object are initialised to `null`. The literal expression reduces to a sequence of field update expressions. The `this` 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 simply looks up the method m in the receiver object and reduces the method body expression e with appropriate values for the parameters $\overline{x}$ and the pseudovariable `this`. It is *only* possible to invoke a method on a *local* object (the receiver's global component ι_a must match that 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 *local* objects.

- 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 expression 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 same actor as the sender) simply appends a new message to the end of the actor's own message queue. The message send itself 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* distinguishes between:

- a regular message m (or the meta-level message 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 $e \leftarrow_f m(\bar{e})$. 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 register$_\mu$, which indicates the registration of a listener function v, to be applied to the value of a resolved future. Since *process* is only invoked on non-future values $\iota_a.\iota_o$, the listener function v is asynchronously applied to $\iota_a.\iota_o$ directly.

- PROCESS-MSG-TO-FUTURE: this rule describes the processing of incoming asynchronous messages directed at local futures. The processing of the message depends on the state of the recipient future, as determined by the auxiliary function *store*. This function returns a tuple (m, e) where `m` denotes either a message or the empty sequence, and e denotes either an asynchronous message send or `null`. The message `m` is then appended to the future's queue, and the actor will continue reducing the expression e. *store* determines whether to store or forward the message `m`, depending on the state of the future and the type of message:
  - If the future is unresolved (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$_\mu$, 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$_\mu$, which indicates a request to register a listener 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$_\mu$, 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 (its value is still ϵ). The paired future is updated such that it is resolved with the value v, and its queue is emptied. The messages previously stored in its queue Q' are forwarded, as described by the auxiliary function fwd. This function generates a sequence of message sends as follows:
  - If the queue is empty, no more messages need to be forwarded and the expression reduces to `null`.
  - If the queue contains a normal message m (or a meta-level message resolve$_\mu$), that message is forwarded to v.

– If the queue contains a meta-level message register$_\mu$, indicating the request to notify the listener function $\iota_a.\iota_o$ when the future becomes resolved, the function is asynchronously applied with the future's resolved value v.

Actor-global reductions We summarize the actor-global reduction rules in Figure 3.2:

- 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 creating actor to communicate further with the newly spawned actor.

- FAR-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression 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*. This function, described further below, returns a set O'' of copied isolate objects that must be added to the recipient's heap and an updated 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 itself evaluates to `null`.

- CONGRUENCE: this rule simply connects the local reduction rules to the global reduction rules.

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

Parameter-passing rules The auxiliary function $pass(\iota_a, O, \overline{v}, \iota'_a)$ 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 originating actor ι_a.

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

The auxiliary function $reach(O, \overline{v})$ returns the set of all isolate objects reachable in O starting from the root values $\overline{v}$. The first two cases define the stop-conditions of this

(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)

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

(NEW-ISOLATE)

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

(INVOKE)

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

(FIELD-ACCESS)

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

(FIELD-UPDATE)

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

(MAKE-FUTURE)

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

(LOCAL-ASYNCHRONOUS-SEND)

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

(PROCESS-MESSAGE)

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

(PROCESS-MSG-TO-FUTURE)

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

(RESOLVE)

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

Figure 4: Actor-local reduction rules.

(NEW-ACTOR)

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

(FAR-ASYNCHRONOUS-SEND)

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

(CONGRUENCE)

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

Figure 5: Actor-global reduction rules.

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 simply defines fresh identities for each isolate in O'. The function *pass* then returns 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.

3.3 Service Discovery

We now extend AT-LITE with the primitives necessary to describe "service discovery", i.e. the ability for objects in different actors to discover one another by means of a publish/subscribe-style mechanism.

We extend AT-LITE actors with a set of exported objects E and a set of import listeners I. We extend values to include types θ. Objects can be exported, and callbacks can be registered, under different types. When the types match, the callback is invoked with the exported object.

We extend the AT-LITE syntax with a means to export objects (export $e\,e$), a means to register callbacks for discovery (discover $e\,e$) and the syntactic sugar whenDiscovered($e \rightarrow x$)$\{e'\}$ to more closely resemble the AmbientTalk `when: type discovered: callback`

Auxiliary functions and predicates

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

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

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

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

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

construct.

Figure 3.3 lists the additional reduction rules for service discovery:

- PUBLISH: to reduce an `export` expression, the first argument must be reduced to a

Extensions for Service Discovery

Semantic Entities

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

Syntax

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

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

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

(PUBLISH)

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

(SUBSCRIBE)

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

(MATCH)

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

Figure 6: Reduction rules for service discovery

type θ 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 E is extended to include the exported object and type. 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 θ 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 listeners I is extended to include the local callback, and the type.

- MATCH: this rule is applicable when a configuration of actors contains both an actor $\iota_{a'}$ that exports an object under a type θ, and a different actor ι_a that has registered a listener under the same type θ. The effect of service discovery is that an asynchronous `apply` message will be sent to the registered listener object in ι_a. The listener 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.

3.4 Robust time-decoupled message transmission

In the calculus presented above, actors are assumed to be permanently connected to all other actors. The real world, however, shows that devices almost always reside in separate networks and only occasionally meet to exchange messages. In this extension to the calculus, we introduce *networks* which completely isolate their actors from other networks but still allow full communication between actors in the same network. Each network has a unique identifier.

Isolating actors from other actors introduces a problem: how can they communicate? Over time actors will move about and join another network, opening up new message transmission opportunities. We formalize this by splitting the message-sending process into two parts: message creation and message transmission. Whenever an actor executes the $\leftarrow$ operator, the message is created and stored in a message outbox (called Q_{out}), to be transmitted at a later stage. This is called *time-decoupled message transmission* [3], as actors don't have to be connected to each other to create asynchronous messages.

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

In the reduction rules, we replace asynchonous message sends (FAR-ASYNCHRONOUS-SEND) by rules for message creation (CREATE-MESSAGE) and message transmission (TRANSMIT-MESSAGE).

Figure 7 lists the additional reduction rules for time-decoupled message transmission:

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

Extensions for time-decoupled message transmission

Semantic Entities

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

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

(CREATE-MESSAGE)

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

(TRANSMIT-MESSAGE)

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

(MOBILITY)

$$\frac{n \neq n'}{K \cup \mathcal{A}\langle \iota_a, O, Q, Q_{out}, n, e \rangle \rightarrow_k K \cup \mathcal{A}\langle \iota_a, O, Q, Q_{out}, n', e \rangle}$$

Figure 7: Reduction rules for time-decoupled message transmission

- 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 networks. Application

of this rule is entirely involuntary, i.e. actors do not themselves choose to move, they are moved around by the system or environment. The precondition ensures that actors do not move to the same network (a no-op).

Time-decoupled messaging weakens the guarantees AT-LITE gives about message ordering. Assume the following scenario: actor A sends a message to actor C at time t_A and an actor B sends a message to the same actor C at a later time t_B (so $t_A < t_B$). Previously, actor C would process the message from A first, then the message from B. With time-decoupling, the ordering depends not on the time of message *creation*, but of message *transmission*. The ordering of messages between any specific pair of actors is still maintained, as messages are still transmitted in a FIFO manner between individual actors.

A further extension could modify the MOBILITY rule to trigger `when:disconnected:` handlers for far references pointing to actors in the old network n. It could then also trigger `when:reconnected:` handlers for far references pointing to actors in the new network n'.

4 Conclusion

We have presented an operational semantics for a key subset of the AmbientTalk programming language. The operational semantics provides a formal account of AmbientTalk actors as communicating event loops, objects, isolates, futures, asynchronous message sends, service discovery and time-decoupled message transmission. To the best of our knowledge, this is the first formal account of an actor language built on the communicating event loops model with non-blocking futures.

References

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

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

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

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

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

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

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

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

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

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