Actors in an Ad-Hoc Wireless Network Environment

Jessie Dedecker[1]* and Dr. Werner Van Belle[2]

[1] jededeck@vub.ac.be
Programming Technology Lab
Department of Informatics
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
Belgium
[2] werner.van.belle@cs.uit.no
Universitetet i Troms/o
Department Computer Science
9021-Norway

Abstract. Today mobile devices can interact with their environment, because of the introduction of wireless communication. Wireless networks have two properties that distinguishes itself from the wired networks: First, wireless communication is less reliable than their wired variants because they have a limited communication range. Second, wireless communication is more dynamic as communication partners enter and leave frequently the communication range of the wireless network. These two distinct properties make the development of mobile software difficult. The actor model is a programming model that allows development of concurrent distributed software in open distributed environments. Actor models fail to handle these two distinct properties of wireless networks well. In this paper we extend the operational semantics of the actor model to capture these two properties in the actor model. We do this by adding a single new concept to the model: the mailbox. This paper provides a foundation for new implementations of the actor language and frameworks that are usable in the context of wireless network environments.

1 Introduction

The introduction of wireless communication technology (such as WiFi, Bluetooth, 802.15.x and others) is a big step towards pervasive computing. Pervasive computing means that the computing technology should be gracefully integrated into our everyday life so that the user is not aware of the technology anymore [1]. To achieve such a goal the software running on mobile devices should adapt to new physical environments as the user walks around. Each physical environment potentially contains other computing devices that can provide the mobile device

* Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

with useful information about the new physical environment. The mobile devices can transparently access the information on these computing devices using wireless communication. The context of pervasive computing also implies that the wireless network environment is ad-hoc. This means that there is no assurance on what services are accessible/present in the physical environment. Hence, in an ad-hoc network environment we cannot rely on central server infrastructure.

Wireless communication is distinct from wired communication, because wireless communication technology has a limited communication range. Therefore, wireless networks exhibit two properties: First, wireless communication is less reliable than their wired variants because connections are dropped as the mobile device is moved out of communication range of the network. Second, wireless communication is more dynamic, because communication partners frequently enter and leave the communication range of the wireless network. These two properties are also present in open distributed network environments, such as the internet, but they are less prominent, because the incidence rate of unavailable communication partners is much lower and communication partners are often unavailable for a small amount of time. Most distributed application frameworks and languages are developed with the assumption of reliable communication, that is to say communication failures are handled as an exception. Communication failures are no longer the exception in wireless communication because, as the mobile devices move and the communication range of the wireless technology is limited, connections are lost and new connections are created. This makes wireless communication more volatile than wired communication. A discrete movement scenario is illustrated in figure 1.

Actors are a programming model for open distributed network environments and provide a good groundwork for programming the internet. However, the actor model fails to capture the high incidence rate of the two properties discussed above. Furthermore, it is difficult to write software in the actor model.

This paper is structured as follows: A brief summary of the actor model and its limitations with respect to wireless network environments is explained in section 3. In section 4 we extend the operational semantics of the actor model and explain the implications of the changes in the context of ad-hoc wireless network environments. In section 5 we show the usefulness of the extensions in two examples. Finally, we conclude the paper (section 7) after we have discussed related work in section 6, but first we explain why asynchronous communication is the appropriate choice to interact in a wireless network environment.

2 Wireless Applications Use Asynchronous Communication

With asynchronous (a.k.a. non-blocking) communication there is no correlation between the time of sending and the time of receiving a message. This decouples the availability of communication partners in time and makes it appropriate in wireless network environments.

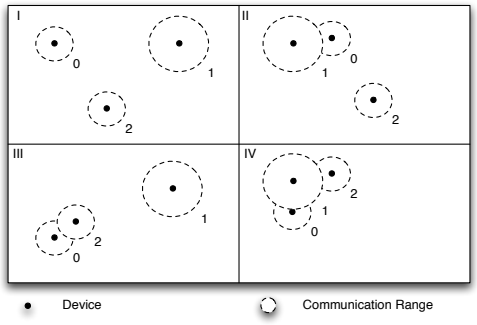

Fig. 1. Discrete Movement Scenario

Distributed applications over wireless communication media often imply a disconnected operation because there can be a very large delay between sending a message and receiving an answer. In extreme cases the delay could be several days or months, for instance when the device is not connected to any network at the moment of sending a message. Therefore, such applications should communicate asynchronously. However, asynchronous communication does not solve all the problems we encounter while developing distributed applications for ad-hoc wireless network environments.

2.1 Computational Context

Asynchronous communication is not a bed of all roses and skittles – it comes with a cost and that cost is the complexity of developing applications. The complexity comes from the fact that computation continues after sending a message. So, after an asynchronous message is sent the actor receives other messages. When the reply to the first message is received the developer needs to restore the computational context in which that message was sent in order to interpret the result it received.

2.2 Environmental Context

Due to the continuous movement of the user from one physical space to another, the available communication partners often change. The devices will often have to postpone tasks it was doing, because a communication has become unreachable. When the user moves again in communication range of a communication partner then it should resume its task, unless of course the task has become obsolete (i.e. because the device was able to do the task at some other place). With physical movement the device starts living in another environmental context with different possibilities (other devices become available) and different limitations (devices disappear).

To simplify the development of distributed applications over wireless technology both contexts should be represented as first-class values [2] by the distributed application framework or language. Now that we have pointed out these requirements we will look at the actor programming language, a distributed programming language that communicates asynchronously.

3 Actors

The actor programming language [3] was designed for use in open distributed network environments (i.e. the internet). A distributed application is modelled with actors that are distributed throughout the network. Each actor has a behavior associated with it. The behavior defines how an actor handles incoming messages. Communication between actors occurs solely with asynchronous message passing.

3.1 The Actor Programming Language

The operational semantics of the actor programming language [4] are defined as an extension of a simple functional language. A function that takes one parameter, a message, and is used to define the behavior of an actor. In the operational semantics of the actor language the functional language is conceptualized by the lambda calculus [5]. The lambda calculus is extended with three actor primitives that support programming in a distributed environment:

- create a new actor using *letactor* primitive. The *letactor* primitive takes one argument. A function that is the initial behavior of that actor.
- send messages to known actors using the *send* primitive. The *send* primitive takes two arguments, the recipient's actor address and a message. Such a message can contain the address of other actors.
- modify its own state and behavior using the *become* primitive. The *become* primitive takes one argument, a function that is the new behavior and state of the actor. There is no shared data between actors.

3.2 Actor Systems

The actor language is supported by an actor system. Conceptually, an *actor system* can be modelled as a message set and the behavior of the actors living in the system. The message set contains two types of messages: 1) messages sent, but not yet transmitted and 2) messages received, but not yet processed. A message, whose target address is that of an actor living in the actor system, is taken from the message set and passed as an argument to the function that is assigned to the target actor address. When the message set is empty the actor waits for a new message. An actor only handles one message at a time. Hence, race conditions can not occur on data-level in the actor language. When a message is sent then the message is put in the message set. When the target

actor address of the message is that of an actor living in another actor system then the message is transferred to the message set of that actor system. The operational semantics of the actor language do not define any order in which the messages from the message set are processed, but it assumes fairness so no starvation can occur.

3.3 Applicability of Actors in Ad-Hoc Wireless Network Environments

The actor model is designed for open distributed network environments, such as the internet, where communication partners are sometimes unavailable for a short period of time. Unavailable communication partners are captured in the actor model through the message sets: messages sent to actors living in an actor system that is unavailable are kept in the message set until the actor system becomes available again for communication. We believe the actor model is also useful in the context of wireless network environments, because of these message sets. Each actor system has its own message set that contains the messages sent, but not yet transmitted to another actor system and the messages received, but not yet processed. The use of message sets has two key advantages with respect to ad-hoc wireless network environments:

- Each actor system is equipped with its own message set. An actor system is therefore self-sufficient and does not rely on a general server infrastructure from its environment for its communication.
- A message set allows asynchronous communication in intermittently connected environments in a transparent way. A message is transferred from one message set to another whenever communication is possible.

There are however some conceptual problems with the actor model, these limitations make it either impossible to use the current actor model in an ad-hoc wireless network environment or difficult to program in the actor language. The limitations are summarized below:

- the model does not support dynamic communication between different actor systems. Actor systems are connected through a static connection.
- the actor model does not define how actor addresses are resolved. The actors needs to know its available communication partners when the user arrives at a new location. We need an abstract way to reference the set of actors the application can communicate with. Currently, the actors have no means to find each other when mobile devices are in communication range of one another.
- the model does not support disconnected operations. The actor model assumes that messages sent will eventually be received. In an ad-hoc wireless network environment this precondition cannot be guaranteed anymore. There is a need for more explicit control over the delivery of messages.
- there is no support for computational/environmental context – the actor model lacks support for the loss of context, as explained in the previous section.

4 The Extended Actor Model

In this section we extend the operational semantics and the syntax of the actor model so that the limitations, pointed out in the previous section, are resolved. We will refrain from giving all the definitions from the original model, which can be found in [4]. We will however repeat some definitions if we adapted them to extend the operational semantics of the actor model or if we believe that they are essential to understanding the extensions.

The main addition to the actor model is the introduction of explicit mailboxes. A mailbox is a store of messages. A number of mailboxes are used within the model to guarantee communication. These are the "in"-box, which keeps track of incoming messages, the "out"-box, which keeps track of messages that should be delivered to other actor systems, the "sent"- and the "rcv"- box which keep track of, respectively, which messages have been sent and which messages have been processed by the actor. Aside from these four standard mailboxes, every actor can create new mailboxes as necessary. Messages can be present in multiple mailboxes at the same time. E.g, a message can be located in the "sent"-box of the sending actor and the "in"-box of the receiving actor.

4.1 Messages and Mailbox Associations

Definition 1 (Messages ($\mathbb{M}$)).

$$\mathbb{M} = \{pr(a, cv) \mid a \in \mathbb{V}, cv \in \mathbb{V}\}$$

A message is a pair (pr) of:

- a, the actor address of the receiving actor.
- cv, a communicable value, constructed from atoms and actor addresses, but not containing lambda abstractions as specified by Agha [4].

$\mathbb{V}$ is the set of all possible value expressions of the functional language as defined in [4]. In the spirit of dynamically typing (as in [4]) we do not restrict the target of the message to the set actor addresses, the correctness is checked in the rules that define the operational semantics. A message is represented as a pair of value expressions, this is in contrast with the message representation as defined in [4] (where a message was denoted with $< a \Leftarrow cv >$). By representing the messages as a pair of values the message becomes a first class value in the actor language. This will proof useful to manipulate the mailboxes. To make a clear distinction in the definitions between messages and "regular" pair values, we will identify a pair that is used as a message with $a \Leftarrow cv$.

In the extended actor model a message is always associated with at least one mailbox. To denote these mailbox associations in the actor model we introduce the following set:

Definition 2 (Mailbox Associations ($\mathbb{mB}$)).

$$\mathbb{mB} = \{< ct \mid mbx_a >\in \mathbb{X} \to \mathbb{S} \to \mathbb{V} \mid ct \in \mathbb{V}, a \in \mathbb{X}, mbx \in \mathbb{S}\}$$

Denotes a content ct associated with mailbox mbx_a. $\mathbb{S}$ is the set of identifiers for mailboxes, $\mathbb{S} \subset \mathbb{A}\mathtt{t}$, with $\mathbb{A}\mathtt{t}$ the set of atoms in the functional language. $\mathbb{X}$ is the set of actor addresses. The mailbox itself is written as an identifier, subscripted with the name of the actor the mailbox belongs to. E.g., in_b denotes the in-box of actor b. Typically, messages are associated with a mailbox, but other values can also be associated with a mailbox.

4.2 Actor Configurations

The operational semantics of the model itself is based on actor-configurations and reduction rules defined on such a configuration. An actor configuration can be considered to be an actor system as explained in section 3.2. Essentially, an actor configuration can be perceived as all actors present on one computational device, such as an embedded system, a desktop and others. The set of actor configurations is defined as

Definition 3 (Actor Configurations ($\mathbb{K}$)).

$$\langle\!\langle \alpha \mid \mu \rangle\!\rangle_{\chi}^{\rho}$$

where ρ, $\chi \in \mathbf{P}_{\omega}[\mathbb{X}], \alpha \in \mathbb{X} \xrightarrow{f} \mathbb{A}\mathtt{s}$, *and* $\mu \in \mathbf{M}_{\omega}[\mathtt{m}\mathbb{B}]$

Say Y is a set then $\mathbf{P}_{\omega}[\mathtt{Y}]$ is the set of finite subsets of Y. $\mathbf{M}_{\omega}[\mathtt{Y}]$ is the set of finite multi-sets with elements in Y. $\mathbb{A}\mathtt{s}$ is the set of actor states as defined in [4].

An actor configuration contains:

- α, the state of the actors in a configuration is given by an actor map α. An actor map is a finite map from actor addresses to actor states. Each actor state is one of
  - $(?_a)$ uninitialized actor state created by an actor named a
  - (b) actor state ready to accept a message where b is its behavior represented by a lambda abstraction
  - $[e]$ actor in a busy state executing expression e. e is either a value expression or a reduction context $\mathtt{R}$ filled with a redex r (written as $\mathtt{R}[r]$). The reduction context is used to identify the subexpression of an expression that is to be evaluated next. For the formal elaboration we refer to the original actor model [4].

  Each mapping of an actor state is subscripted by their actor address. E.g. $(?_a)_c$ denoted that uninitialized actor c that was created by actor a.
- μ, a multi-set of mailbox associations.
- ρ, receptionists, the actors from this configuration that are remotely accessible from other actor configurations
- χ, external actors, the references to remote actors from other actor configurations that can be accessed from this actor configuration.

It is required that all actor configurations satisfy the following constraints (A=Dom(α)):

1. $\rho \subseteq A$ and $A \cap \chi = \emptyset$,
2. if $\alpha(a) = (?_{a'})$, then $a' \in A$,
3. if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$,
4. if $< ct \mid mbx_a > \in \mu$, then $a \in A$
5. if $ct = v_0 \Leftarrow v_1$ then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$

The fourth point is new compared to the constraints that were defined in [4] and denotes that each mailbox in an actor configuration should be owned by an actor from the actor configuration.

4.3 Operational Semantics of Actor Configurations

On such an actor configuration, a number of reduction rules are defined. Each rule contains a label l that consists of a tag indicating the name of the primitive instruction and a set of parameters. In all cases, except for the i/o transitions (with tags *in*, *out*, *join*, *disjoin*), the first parameter names the $focus$ actor of the transition. As in, [4] if $\alpha'(a) = (b)$ and is with a omitted from its domain we write α' as $\alpha, (b)_a$ to focus attention on a. We follow a similar convention for other states subscripted with addresses (such as mailbox associations).

The rules are in our model are extended with an environmental set τ. The set τ contains the actor configurations that are available (in communication range) while the reduction is performed. The introduction of this set is important to reify the notion of environmental context in our extended model. Below we explain and discuss the different rules.

Definition 4 ($\underset{\tau}{\mapsto}$)**.** $\tau \in \mathbf{M}_\omega[\mathbb{K}]$

$< \texttt{fun} : \texttt{a} >$

$$e \overset{\lambda}{\mapsto} Dom(\alpha) \cup \{a\}\ e' \Rightarrow \langle\!\langle \alpha, [e]_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, [e']_a \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$< \texttt{new} : \texttt{a}, \texttt{a}' >$

$$\langle\!\langle \alpha, \texttt{[R[newadr()]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[a']]}_a, (?_a)_{a'} \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$< \texttt{init} : \texttt{a}, \texttt{a}' >$

$$\langle\!\langle \alpha, \texttt{[R[init(a', v)]]}_a, (?_a)_{a'} \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_a, v_{a'} \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$< \texttt{bec} : \texttt{a}, \texttt{a}' >$

$$\langle\!\langle \alpha, \texttt{[R[become(v)]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_{a'}, v_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \qquad a' fresh$$

$< \texttt{send} : \texttt{a}, \texttt{m} >$

$$\langle\!\langle \alpha, \texttt{[R[send}(v_0, v_1)\texttt{]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_a \mid \mu, M \rangle\!\rangle^{\rho}_{\chi}$$
$$with\ M = \{< v_0 \Leftarrow v_1 \mid out_a >\}\ iff\ v_0 \notin Dom(\alpha)$$
$$or\ M = \{< v_0 \Leftarrow v_1 \mid sent_a >, < v_0 \Leftarrow v_1 \mid in_{v_0} >\}\ iff\ v_0 \in Dom(\alpha)$$

$< \texttt{out} : \texttt{m} >$

$\langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu, m' \rangle\!\rangle^{\rho \cup (FV(cv) \cap Dom(\alpha))}_{\chi}$
with $m =< b \Leftarrow cv \mid out_a >$
and $m' =< b \Leftarrow cv \mid sent_a >$, $b \in \chi, a \in Dom(\alpha)$
if $\exists \kappa \in \tau$ *with* $\kappa = \langle\!\langle \alpha_1 \mid \mu_1 \rangle\!\rangle^{\rho_1}_{\chi_1}$ *and* $b \in Dom(\alpha_1)$

$< \texttt{in} : \texttt{m} >$

$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi \cup (FV(cv) - Dom(\alpha))}$
with $m =< b \Leftarrow cv \mid in_b >$, $b \in \rho$ *and* $FV(cv) \cap Dom(\alpha) \subseteq \rho$

$< \texttt{rcv} : \texttt{a}, \texttt{m} >$

$\langle\!\langle \alpha, (v)_a \mid \mu, m \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[app(}v, a \Leftarrow cv\texttt{)]}_a \mid \mu, m' \rangle\!\rangle^{\rho}_{\chi}$
with $m =< a \Leftarrow cv \mid in_a >$ *and* $m' =< a \Leftarrow cv \mid rcv_a >$

$< \texttt{messages} : \texttt{a}, \texttt{mbx} >$

$\langle\!\langle \alpha, \texttt{[R[messages(mbx)]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[}(ct_1, \ldots, ct_n)\texttt{]]} \mid \mu \rangle\!\rangle^{\rho}_{\chi}$
with $ct_i \in \{ct \mid < ct \mid mbx_a >\in \mu\}$

$< \texttt{add} : \texttt{a}, \texttt{mbx}, \texttt{m} >$

$\langle\!\langle \alpha, \texttt{[R[add(mbx, ct)]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_a \mid \mu, m \rangle\!\rangle^{\rho}_{\chi}$
with $m =< ct \mid mbx_a >$

$< \texttt{del} : \texttt{a}, \texttt{mbx}, \texttt{m} >$

$\langle\!\langle \alpha, \texttt{[R[del(mbx, ct)]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_a \mid \mu' \rangle\!\rangle^{\rho}_{\chi}$
with $\mu' = \mu \backslash \{< ct \mid mbx_a >\}$

$< \texttt{join} >$

$\langle\!\langle \alpha_0 \mid \mu_0 \rangle\!\rangle^{\rho_0}_{\chi_0} \underset{\tau}{\mapsto} \langle\!\langle \alpha_0 \mid \mu_0, M \rangle\!\rangle^{\rho_0}_{\chi_0}$
if $\exists \kappa \in \tau$ *with* $\kappa = \langle\!\langle \alpha_1 \mid \mu_1 \rangle\!\rangle^{\rho_1}_{\chi_1}$ *and*
$M = \{< a \Leftarrow cv \mid joined_b >, < b \Leftarrow \texttt{join} \mid in_b > \mid < cv \mid required_b >\in \mu_0 \wedge <$
$cv \mid provided_a >\in \mu_1\}$

$< \texttt{disjoin} >$

$\langle\!\langle \alpha_0 \mid \mu_0 \rangle\!\rangle^{\rho_0}_{\chi_0} \underset{\tau}{\mapsto} \langle\!\langle \alpha_0 \mid \mu_0 \backslash T, M \rangle\!\rangle^{\rho_0}_{\chi_0}$
if $\nexists \kappa \in \tau$ *with* $\kappa = \langle\!\langle \alpha_1 \mid \mu_1 \rangle\!\rangle^{\rho_1}_{\chi_1}$ *and*
with $M = \{< a \Leftarrow cv \mid disjoined_b >, < b \Leftarrow \texttt{disjoin} \mid in_b > \mid$
$< a \Leftarrow cv \mid joined_b >\in \mu_0 \wedge a \notin Dom(\alpha_1)\}$
$T = \{< a \Leftarrow cv \mid joined_b > \mid < a \Leftarrow cv \mid joined_b >\in \mu_0 \wedge a \notin Dom(\alpha_1)\}$

Basic Actor Operations The first four reduction rules below are the same as defined in [4] and did not need to be adapted:

- The $< fun >$ rule above delegates the purely functional expressions used in the actor program to the functional redexes. The functional redex contains reduction rules for function calls, cons-cell manipulation, branch-testing, type-testing and equality. For the exact definition of these reduction rules we refer the reader to [4].
- The $< new >$ rule is used to create a new actor with address *a'*. The new actor is not initialized after the *new* reduction. The new uninitialized actor is denoted with $(?_a)_{a'}$.
- With the $< init >$ rule a new actor is initialized with behavior v. Only the actor that created the actor *a'* can initialize it.
- With the $< bec >$ rule the actor can change its state and its behavior. The actor address *a'* is anonymous and thus unknown to all other actors. We say that a variable is *fresh* with respect to a context of use if it does not occur free or bound in any syntactic entity.

Adapted Rules The remainder of the rules have been adapted to include the notion of mailboxes:

- The $< send >$ rule differs from the original *send* rule. Instead of placing the message immediately in the message set μ, we first differentiate on the nature of the message. If the message can be delivered locally (within the same actor configuration), it is immediatelly placed in the target its in-box, and the sent-box of the sender. Otherwise, the message is placed in the actor its out-box. From there on, through other reduction rules, this message will be moved to the target its in-box.
- $< out >$ We write FV(e) for the set of free variables of e. Messages that cannot be delivered locally, will be placed in the out-box of the sending actor. The out reduction rule is used, at the sending side, to transmit a message to another actor configuration. The rule explicitly states, through the environmental set τ, that the actor configuration should be in communication range with another actor configuration that contains the target actor of the message. Then the outgoing message will be removed from the out-box and placed in the sent-box. This allows the actor to verify which messages have actually been sent. Similar to the original model, the set of receptionists is expanded with the local actor addresses that were communicated in the message.
- $< in >$ This rule is called when an actor configuration receives a message from an external actor that lives on another actor configuration that simulataneously will be performing the rule with tag *out*. In this situation, the message is placed in the in-box of the target actor.
- $< rcv >$ When a message is available in the in-box of an actor, it can be received by the actor. Once the message is being processed by the actor, the message is moved to the *rcv* mailbox. This means that an actor has a

history of the messages that it processed. This proves to be a useful session management utility as is shown in the examples in section 5.

Mailbox Manipulation $< messages >, < add >, < del >$ Aside from these modified standard rules, some reduction rules have been added to manipulate and inspect the mailboxes from within the actor language. With the $< messages >$ rule one can access the content of a mailbox. The $< add >$ rule will create a mailbox when it does not exist, if the mailbox exists, the content will be added to the mailbox. The $< del >$ rule will delete a message from a mailbox, when the last message of a mailbox has been removed, the mailbox itself is removed. The above reduction rules allow actors to manage mailboxes explicitly. It also allows actors to keep track of a history of sent and waiting messages. It allows actors to see whether a message has actually been sent out. Also note that there is no rule in which a message automatically disappears from the system. This means that memory management will have to be handled manually by the programmer. This is because it depends on the semantics of the program whether a message has become irrelevant to the program. For example, when a certain task has completed and is not relevant anymore.

Handling Environmental Contexts Mobile applications used in an ad-hoc wireless network environment need to "sense" their physical environment. This "sense" is introduced in the model through the notion of first-class "environmental context". The first-class "environmental context" is supported with two reduction rules: $< join >$ and $< disjoin >$. When two devices come in contact (because they are in the same communication area), they will automatically "join". They disjoin when they leave each others communication range. An important issue with the "join" is the abstract specification of actors we are looking for in the physical environment, since the physical environment is possibly unknown we do not know the addresses of the actors that are living on other machines. To this end, we have added four extra mailboxes for every actor: *provided*, *required*, *joined* and *disjoined*. The mailboxes *provided* and *required* are used to let an actor specify an abstract description of what kind of behavior it provides or requires, this abstract description is called a *pattern*. The pattern is specified in the model as a communicable value. When a pattern in the *provided* and *required* mailboxes match, then the corresponding actors will be notified. This notification happens through the use of the *joined* and *disjoined* mailboxes. Thus, the *joined* and *disjoined* mailboxes keep track of the relevant actors, specified through the use of the *provided* and *required* mailboxes, that are in communication range. This mechanism is defined in the model through the $< join >$ and $< disjoin >$ rules:

- $< join >$ when two actor configuration come in communication range. Every actor b that requires a certain pattern cv, which has become available in another actor configuration κ that is in communication range, will be informed of this by receiving a "join" message in its in-box. Also, for every matching

pair of required-provided patterns, the corresponding *joined* mailbox is updated. In the *joined* mailbox, a special kind of messages is stored, namely a `resolution`. A resolution contains a) the kind of `pattern` (*cv*) that has been matched and b) a `provider` a who provides the service.
- $< disjoin >$ when two actor configurations leave each others communication range. Every actor that is aware of another joined actor, that has become unavailable, will be informed of the disjoin. Once an actor is informed the corresponding resolution is removed from the joined mailbox. Actors that have removed the matching messages from their *joined* mailbox will not be informed.

The join and disjoin operations are not symmetrical. After joining and disjoining two actor configurations, the state of the involved actor configurations is not necessarily the same as before the join operation. This is due to the fact that for every join or disjoin a number of messages are sent, which might influence the behavior of the involved actors.

5 Examples

Now that we have defined the operational semantics of the extended actor model we show that it is useful in the context of ad-hoc wireless network environments by means of two examples. The examples are defined with actor code using the extended actor model from the previous section. The first example shows how anonymous communication can be expressed. In the second example we work out a meeting scheduler application for use in an ad-hoc wireless network environment.

5.1 Pattern-Based Communication

We often want to send messages to an actor providing some service, but the actor address is unknown at the time of message sending, because the actor address depends on the physical environment of the embedded device. The anonymous actor can be described using type information [6] or more semantic information about the service (such as a QoS property of the communication partner). In this example, we show how we can express such anonymous communication using the mailboxes in the extended actor model. We define a new communication primitive *psend* that takes two parameters: a description *pattern* of the required actor and the message that needs to be send.

```
psend = λpattern.λmsg.
   seq(add(required, pattern),
       add(pending, msg))
```

We add the description *pattern* of the required actor to the *required* mailbox. This way the actor will be notified when the actor configuration joins with another

actor configuration providing this pattern. The *pending* mailbox contains the messages that have a pattern as destination instead of an actor address.

The *handleJoin* definition walks over the resolutions in the *joined* mailbox. Each time a pattern that corresponds with the target of the messages in *pending* mailbox is found, the message is send to the provider of the pattern and removed from the *pending* mailbox.

```
handleJoin = λm.
    if(join?(m),
       for-each(λresolution.
           for-each(λmsg.
               if(eq?(target(msg), pattern(resolution)),
                  seq(send(provider(resolution), msg),
                      delete(pending, msg))),
               messages(pending)),
           messages(joined)))
```

Below is the definition of an actor that wants to print a file from the moment it comes into communication range of an actor that provides a printing service.

```
B_Customer = λfile.λm.
    seq(psend('printer@300dpi, mkPrint(file)),
        become(handleJoin(m)))
```

This example shows that the extended actor model contains the basic primitives to build more complex discovery mechanisms tailored to the need of the application.

5.2 Meeting Scheduler

Suppose a group of people all walk around with a PDA that is equipped with a wireless transmitter. Each PDA has an agenda application running and the agenda can be used to schedule a meeting with a group of acquaintances. A request for a meeting can be made at any point of time (so the acquaintances do not have to be reachable at the point of requesting a meeting). We assume that the PDA's at some point in time will be in communication range. The owner of the PDA's does not have to be aware of the communication that goes on between the PDA's, so the PDA can become out of range at any point in the communication. The agenda application schedules a meeting in two steps:

1. try to make a reservation in the agenda from the participants of the meeting
2. confirm the meeting in the agenda from the participant if all were successfully reserved.

If the reservation fails at some point, then all reservations that were made on other agenda's are removed. Our meeting scheduler is modelled by two actors that are described below.

Agenda Actor The agenda is initialized with the e-mail address of the owner from the agenda. The e-mail address is used together with the type information of the application to join actor configurations together.

```
B_InitAgenda = λemail.λm.
   seq(add(provided, mkProvided(email)),
       become(B_FreeAgenda()))
```

We have chosen to represent the agenda as a single slot that is available for meetings to reduce the length of our example. The slot has three states: *free*, *reserved* and *confirmed*. The agenda understands three messages:

- *free*: when this message is received and the slot is *reserved* then it becomes available for reservation. This message is used to undo a reservation.
- *reserve*: when this message is received and the slot is *free*, then the slot becomes *reserved*.
- *confirm*: when this message is received and the slot is *reserved* then the slot becomes *confirmed*.

The slot contains an id that is used to determine to whom the slot has been assigned when it is *reserved* or *confirmed*. The slot only evolves into the corresponding state if the message contains the right id.

The code below shows the implementation of the agenda. The function *rec* in the code calculates the fixed-point of a function [7] so that it can be called recursively.

```
B_FreeAgenda = rec(λb.λm.
    if(free?(m),
       become(b()))
    if(reserve?(m),
       seq(send(sender(m), mkReserveAnswer(session(m), #true)),
           become(B_ReservedAgenda(session(m))))))

B_ReservedAgenda = rec(λb.λid.λm.
    if(and(free?(m), eq?(id, session(m))),
       become(B_FreeAgenda()))
    if(reserve?(m),
       seq(send(sender(m), mkReserveAnswer(session(m), #false)),
           become(b(id))))
    if(and(confirm?(m), eq?(id, session(m))),
       become(B_ConfirmedAgenda(id))))

B_ConfirmedAgenda = rec(λb.λid.λm.
    if(reserve?(m),
       seq(send(sender(m), mkReserveAnswer(session(m), #false)),
           become(b(id)))))
```

Scheduler Actor The scheduler agent is responsible for contacting the agenda actors to schedule the meeting. In the scheduler actor definitions below we use three helper functions:

- a *filter* function that uses a predicate to filter elements in a list[3]

```
filter = rec(λb.λpredicate.λlist.
  if(empty?(list),
     emptyList,
     if(predicate(car(list)),
        cons(car(list), b(predicate, cdr(list))),
        b(predicate, cdr(list)))))
```

- a *map* function that returns a transformed list and takes two parameters: a function that transforms elements and a list that is to be transformed is defined as in most standard Scheme implementations [8].
- *msend* that allows to send a message to a list of actor addresses or actors described with pattern descriptions (such as in the previous example)

```
msend = λtargets.λm.
  for-each(λtarget.
              if(actorAddress?(target),
                 send(target, m),
                 psend(target, m)),
           targets)
```

- *madd* that allows to add a list of messages to a mailbox

```
madd = λmbx.λitem.for-each(λitem.add(mbx,item), items)
```

The scheduling agent is initialized as $B_{InitScheduleAgent}$. The scheduler agent has an id that is used to identify its session.

$B_{InitScheduleAgent}$ `= rec(λb.λid.λm.`

```
    if(schedule?(m),
       seq(madd(required,
                map(λemail.mkRequired(email), partipants(m))),
           msend(participants(m), mkReserve(id)),
           become(B_ReserveScheduleAgent(id, participants(m), sender(m)))))
```

The schedule agent can be requested to schedule a meeting by sending it the message *schedule*. When such a request is received the agent sends out *reserve* messages to the agenda actors of the participants and the scheduler evolves into the $B_{ReserveScheduleAgent}$ state.

[3] In the λ calculus a pair is created with the function *cons*. The function *car* returns the first element of the pair, while the *cdr* function returns the second element. A list of elements is represented as a nesting of pairs. e.g. (1, 2, 3) is represented as (1, (2, (3, '())))

```
B_ReserveScheduleAgent = rec(λb.λid.λparticipants.λm.λcustomer.
    if(and(reserveAnswer?(m), eq?(id, session(m))),
       if(success?(m),
          if(eq?(map(sender, filter(reserveAnswer?, messages(rcv)))),
                     participants),
             seq(msend(participants,mkConfirm(id)),
                 become(B_ConfirmScheduleAgent(id, participants, customer)))),
          seq(msend(map(destination, filter(reserve?, messages(sent)))),
                    mkFree(id)),
              mdelete(out,
                     map(identity, filter(reserve?, messages(out)))),
              send(customer, mkFailed()),
              become(B_InitScheduleAgent(id+1)))))))
```

When handling a *reserveAnswer* message we make use of the mailboxes introduced in our system:

- If the reservation was successful we check the box of received messages (named *rcv*) to see if we have received an answer from all the participants their agenda's. Due to the mailboxes there is no need to manually maintain the sessions. If all agenda's are successfully reserved, then the *ScheduleAgent* actor sends *confirm* messages to all Agenda's.
- If the reservation fails at some point we have to free up the agenda's of the reservations that were succesfull. We can use the mailbox of the messages that were sent (named *sent*) to track to which agenda actors we already sent out reservation request. To these actors we send the *free* message so that they can undo their reservation. We then delete the *reservation* messages that were not sent out from the mailbox *out*. We also notify the customer actor that sent the *schedule* message that the meeting could not be scheduled by sending it a *failed* message.

```
B_ConfirmScheduleAgent = rec(λb.λparticipants.λm.λcustomer.
    if(and(disjoin?(),
           eq?(map(sender, filter(confirm?, messages(sent)))),
               participants))
       seq(send(customer, mkSucceeded()),
           become(B_InitScheduleAgent(id+1))))
```

Each time the *ScheduleAgent* actor disjoins from an agenda actor it checks to see if all *confirm* messages were sent out, again using the mailbox *sent*. If all *confirm* messages were sent, then the customer actor that sent the *schedule* is notified with a *succeeded* message.

The second example shows that the mailboxes introduced in the extended actor model contain useful primitives that allow to structure the different conversations the scheduler has with the different agenda applications. It also shows the usefulness of the mailboxes "sent" and "out" to check and manipulate the communication status of different messages.

6 Related Work

6.1 Formalisms

The distributed join calculus [9] gives chemical semantics for logical movement (migration), failures and failure detection of agents. A disadvantage of the model is that it presumes that failures can be detected. In the context of ad-hoc wireless network environments failures are impossible to detect, because it is technically impossible to know whether a device has failed or the device has moved out of communication range.

Locality based linda [10] is a formalisation of Linda [11, 12] (discussed below) but with the explicit notion of locations. Locations are added to the primitives of Linda, so the primitives work on a specific tuple space. Anonymous communication becomes impossible, because the location needs to be provided. In our context this would mean that the addresses of communicating devices need to be known, which is impossible to determine in ad-hoc wireless network environments. The model also does not define what happens with communication if the location is unavailable.

The mobile ambient calculus [13] expresses an abstraction between mobile computing and mobile computation in administrative domains. Mobile computation is the physical movement of devices between administrative domains, while mobile computing is the movement of a logical process (also called migration). The extended actor model and the ambient calculus are complementary. The mobile ambient calculus describes movement of processes between different locations and communication between ambients that are located in an ambient. The extended actor model describes how actors can structure their asynchronous communication to manage sessions and how new actors can be found in new physical environments. The mobile ambient calculus could be used to describe the physical movement of actor configurations.

6.2 Software Technology

The programming language Janus [14] is based on concurrent constraint logic programming (CCP). We learned about it shortly after writing this paper. The model also includes explicit mailboxes and they show that the use of mailboxes is more expressive than the original actor model. Agents communicate by putting messages in each others mailboxes. A mailbox is associated with a "teller" and an "asker". The teller allows agents to put messages in the mailbox and the asker allows agents to retrieve messages from the mailbox. Tellers and askers can be transmitted between agents. An agent can ask for input on one or more mailboxes at the same time. The Janus programming model can express useful distributed programming idioms. There are some differences however between our extended model and theirs. The mailboxes are used as a communication medium in Janus, while in our extended actor model they are used for three different purposes: 1) a means to track the communication history 2) the actors that are communication range and 3) as a structuring tool for session management.

JXTA [15] is middleware for peer-to-peer environments and covers problems such as peer discovery and peer-to-peer communication. JXTA can also be employed in a wireless context [16]. A disadvantage however is that wireless devices have to depend on JXTA relays in order to perform basic operations such as peer discovery. This means that JXTA relays have to be present in each physical environment before wireless devices are able to interact with one another. This is in conflict with the idea of transparency in pervasive computing.

Jini [17] enables discovery and lookup of services based on their description. It is possible to search for objects based on the object's type information and attributes. Jini is based on a central server architecture. A Jini server would have to be present in any place where devices need to find each other.

In [6] a library is introduced to enable multicast and broadcast messages between objects in Java using wireless communication. Message delivery is based on the type information instead of addresses. The mechanism of sending messages based on type information is similar to our dynamic join that uses a pattern to find compatible devices. However, the lookup mechanism provided by the dynamic join in our extended actor model allows the use of other useful static information such as the e-mail address we used in the example of the meeting scheduler. This can simplify lookup of specific devices without having to know the exact actor address. Messages are also sent asynchronously in the library, but there is no notion of an outgoing queue. This means that if a message is sent and there is no device listening on the other side, then the message is lost. This means that checking message delivery is put on the shoulders of the developer, which will make the program inherently more complex.

Linda [11, 12] is a coordination language based on tuple spaces. There are several primitives that make communication explicit in the code. There is a primitive for putting values in the tuple space (asynchronous) and primitives for getting values from the tuple space (blocking). Tuple spaces are interesting, because they allow communication without the need to specify the communication address of the receiver, much like in the spirit of our join rule. Linda is based on a central server architecture, there is one central tuple space where the structured values are put, which makes it not suitable for ad-hoc wireless network environments. The language does not provide semantics of communication failures in the language, which is important in our context.

7 Conclusion

In this paper we extended the operational semantics of the actor model in order to deal with three problems associated with ad-hoc wireless network environments:

1. How do several devices find each other?
   To handle this problem we introduced reduction rules *join and disjoin* that define what happens when devices come in communication range of one another.

2. How to deal with communication failures and disconnected operation?
   This problem is handled with the introduction of the *in*, *rcv*, *out* and *sent* mailboxes that allow us to track and intervene in the communication between different devices.
3. How to keep track of different conversations that occur in different places?
   This problem is handled through the custom manipulation of mailboxes.

These three problems are handled with the introduction of a single concept in the actor language, namely that of a mailbox. The mailbox unifies the first-class computational and environmental context into a single concept. In the two examples above we have illustrated that they can be used as a basis for the implementation of more complex applications. We have implemented our extended actor model and are looking into new programming constructs based on the extended actor model that will ease the implementation of distributed applications for ad-hoc wireless network environments.

Acknowledgements

Thanks to Thomas Cleenewerck for proof-reading this paper and for providing us with some useful hints to increase the readability of this paper.

References

1. Weiner, M.: The computer for the 21st century. Scientific American **265** (1991) 66–75
2. Keays, R., Rakotonirainy, A.: Context-oriented programming. In: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, ACM Press (2003) 9–16
3. Agha, G.: Concurrent object-oriented programming. Communications of the ACM **33** (1990) 125–141
4. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7** (1997) 1–72
5. Church, A.: The Calculi of Lambda-Conversion. Volume 6 of Annals of Mathematical Studies. Princeton University Press, Princeton (1985)
6. lan Kaminsky, Bischof, H.P.: Many-to-many invocation: A new object oriented paradigm for ad hoc collaborative systems. 17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002) (2002)
7. Mason, I., Talcott, C.: Equivalence in functional languages with effects. Journal of Functional Programming **1** (1991) 287–328
8. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation. MIT Press, Cambridge, MA (1985)
9. Fournet, C., Gonthier, G., Levy, J.J., Maranget, L., Remy, D.: A Calculus of Mobile Agents. In Montanari, U., Sassone, V., eds.: Proceedings of 7th International Conference on Concurrency Theory (CONCUR'96). Volume 1119 of LNCS. Springer-Verlag, Berlin, Germany (1996) 406–421

10. De Nicola, R., Ferrari, G., Pugliese, R.: Locality based Linda: Programming with explicit localities. Lecture Notes in Computer Science **1214** (1997) 712–??
11. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112
12. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM **32** (1989) 444–458
13. Cardelli, L., Gordon, A.D.: Mobile ambients. In Gordon, A., Pitts, A., Talcott, C., eds.: Electronic Notes in Theoretical Computer Science. Volume 10., Elsevier (2000)
14. Kahn, K., Saraswat, V.A.: Actors as a special case of concurrent constraint (logic) programming. In: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, ACM Press (1990) 57–66
15. Gong, L.: Project jxta: A technology overview. Technical report, SUN Microsystems, http://www.jxta.org/project/www/docs/TechOverview.pdf (2001)
16. Gong, L.: Jxta for j2me extending the reach of wireless with jxta technology. Technical report, SUN Microsystems, http://www.jxta.org/project/www/docs/JXTA4J2ME.pdf (2002)
17. Arnold, K., Wollrath, A., O'Sullivan, B., Scheifler, R., Waldo, J.: The Jini specification. Addison-Wesley, Reading, MA, USA (1999)