Mobile Objects
(Preliminary Report)

Michele Bugliesi
Dipartimento di Informatica
Università Ca' Foscari di Venezia
michele@dsi.unive.it
http://www.dsi.unive.it/~michele

Giuseppe Castagna
C.N.R.S.
Laboratoire d'Informatique, ENS
Giuseppe.Castagna@ens.fr
http://www.dmi.ens.fr/~castagna

Abstract

We describe a general model for integrating object-oriented features in calculi of mobile agents. We then study two complementary instances of the model based, respectively, on subjective and objective moves: the interest in the resulting calculi is illustrated by means of a number of examples. The work we describe is not finished: this report should be understood as a first step towards defining a computation model for distributed applications, where conventional technology – based on remote exchange of messages between static sites – and mobile agents coexist and can be integrated in a uniform way.

1 Introduction

Calculi of mobile agents are receiving increasing interest in the programming language community as advances in computer communications and hardware enhance the development of large scale distributed programming. Independently of the new trends in communication technology, object-oriented programming has established itself as the de-facto standard for a principled design of complex software systems.

In this paper, we introduce a general model for integrating object-oriented features into calculi of mobile agents. The model results from extending agents with method definitions and constructs for dealing with method invocations and self denotations. We study several possible characterizations of these mechanisms and discuss their impact on the model.

We then study two complementary instances of the model, that we name *Ambjects* and *Sealjects*. The two instances are based respectively on *Mobile Ambients* [3, 5] and the *Seal Calculus* [19], and they inherit the central features of these two calculi: as a consequence, Ambjects and Sealjects differ in the way they handle names and mobility. Ambjects are based on what is referred to as a "subjective" approach whereby the name and the moves of an agent (an ambject) are decided by the agent itself (its controlling process). Sealjects, instead, are based on an "objective" model in which names and moves of an agent (a sealject) are controlled by the agent's enclosing environment. Also, Sealjects are endowed with stronger security mechanisms than ambjects: specifically, the calculus does not allow the boundary of an agent to dissolve and its content to merge with the enclosing environment.

We illustrate the calculi of Ambjects and Sealjects with a number of rather variated examples. For Ambjects we show how it is possible to encode primitives like method extension and method overriding distinctive of object calculi; we also show that a form of asynchronous channel-based communication can be encoded. For Sealjects, instead, we show how they can be used to specify a subset of Java and we hint at the specification of *ProActive* [7], a Java library for concurrent and distributed programming. Some familiarity with Ambients and Seals is assumed throughout: the reader may refer to [3, 5, 19] for more details.

The interest in the general model and its instances is not in their expressive power: although at the time of writing we have not investigated this issue in depth, it is certainly the case that the extensions can be encoded in the original calculi. Instead, we seek for a principled approach to structuring agents: embedding methods into agents, rather than encoding them, has at least two appealing consequences. Firstly it allows us to define the interface of an agent and its type in a quite precise way. Secondly, it opens the way to reusing the advances in type system of object-oriented programming and static analysis. This will be one of the natural continuations of this work.

Several proposals of formalisms for foundational study of concurrent object-oriented programming can be found in the literature (see for example [11, 13, 16, 17, 20]). However, to our knowledge, this is the first attempt to directly embed methods in calculi of mobile agents.

2 Mobile Objects

Our model of mobile objects results from extending calculi of mobile agents over distributed systems, endowing agents with methods. The definition of the model is largely independent of the underlying calculus of processes and agents. For this calculus we only presuppose the existence of the following constructs: $\mathbf{0}$, denoting the inert process, $P \mid Q$, denoting the parallel composition of two processes P and Q, $x[P]$, denoting the process (or agent) named x running the process P, $(\boldsymbol{\nu}\, x)P$, that restricts the name x to P, and finally $\alpha\,.\,P$, that performs the action α and then continues as P. Clearly, the actions will include at least primitives for moving agents over the locations of the distributed system, but we may disregard those primitives at this stage. What instead is central to the model is the naming mechanism for processes: named processes are abstractions of both agents and, since naming allows nesting, of hierarchically structured locations [3]. So they model both the nodes of the distributed system and the agents over it.

Our proposal is to enrich agents with methods by generalizing their definition as in $x[M;P]$, where P is a process and M a set (rather, a list) of method definitions.

2.1 Syntax and Structural Congruence

The syntax of agents is defined by the productions in Figure 1.

Processes				Methods			
P	$::=$	$\mathbf{0}$	inactivity	M	$::=$	$m(\vec{x}) \triangleright P$	method
	$\mid$	$P \mid P$	parallel composition		$\mid$	$M;M$	method sequence
	$\mid$	$x[M;P]$	ambject or sealject		$\mid$	ε	empty sequence
	$\mid$	$(\nu x)P$	restriction				
	$\mid$	$\alpha.P$	action				

Figure 1: Syntax of Agents

In the following we use P, Q, R to range over processes, M, N to range over (possibly empty) method sequences, x, y to range over generic names, while we use m to single out names in message position. We also omit trailing or isolated $\mathbf{0}$ processes, using α and $x[M;]$ to abbreviate $\alpha\,.\,\mathbf{0}$ and $x[M;\mathbf{0}]$ respectively. Similarly we omit the leading semicolon in case of empty method sequence,[1] writing $x[P]$ rather than $x[;P]$. Finally, we use the metavariable B to denote generic agent bodies and write $x[B]$ rather than $x[M;P]$.

A method definition has the form $m(\vec{x}) \triangleright P$ where the label m is the name of the method, $\vec{x}$ (a shorthand for $x_1, \ldots, x_n$) the list of its formal parameters, and P its body. Instead of the ς-binder of the Object Calculi of [1], we use the keyword *self* to denote the host agent within the body of the agent's

[1] The correct definition of method sequence should be $M ::= \varepsilon \mid m(\vec{x}) \triangleright PW \quad W ::= \varepsilon \mid ;m(\vec{x}) \triangleright PW$. This definition avoids the case of two consecutive trailing semicolons or of a leading one, but it is less readable than the one in Figure 1

methods[2]. Since agents are named, explicit names could be used to invoke methods from any agent, including sibling methods from the same agent: however, as we shall see, the *late binding* semantics associated with *self* ensures a smooth and elegant integration of method invocation and agent mobility.

The syntax (as well as the semantics) could be rephrased to use ς-binders exactly as in [1]: the use of a keyword is simply a notational convenience (see however Footnote 3). A more interesting aspect of the syntax of methods is that, unlike [1], in our model method names are ordinary names, which therefore can be restricted, to define local (or *private*) methods as m in $(\boldsymbol{\nu}\, m)x[m(\vec{y}) \triangleright P; M; Q]$, or communicated as values. Other object calculi in which messages are first class have been studied before in the literature on objects and object type systems: see for instance [8, 15, 2].

To complete the presentation of the syntax, we are now left with defining the structure of actions. As we shall discuss later, the format (and associated behavior) of actions is what distinguishes ambjects from sealjects. For the time being, we will focus on the only action which pertains to both the the two calculi, namely, message sends. Message sends are denoted by $x \leftarrow m(\vec{y})$, where x is the name of an agent, and $m(\vec{y})$ invokes the m method found in x with arguments $\vec{y}$. Unlike [1], the format of a message send requires that the recipient be the *name* of an agent rather than the agent (the object, in [1]) itself. The semantics of message sends is discussed in detail in Section 2.2. Before that, we define the relation of structural congruence for agents.

$$
\begin{array}{lrl}
P \mid \mathbf{0} \equiv P & & \text{(Struct Dead Par)} \\
P \mid Q \equiv Q \mid P & & \text{(Struct Par Comm)} \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & & \text{(Struct Par Assoc)} \\
(\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} & & \text{(Struct Res Dead)} \\
(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P & \text{for } x \neq y & \text{(Struct Res Res)} \\
(\boldsymbol{\nu}\, x)(P|Q) \equiv P|(\boldsymbol{\nu}\, x)Q & \text{for } x \notin \mathit{fn}(P) & \text{(Struct Res Par)} \\
 & & \\
(\boldsymbol{\nu}\, x)y[M;P] \equiv x[M;(\boldsymbol{\nu}\, x)P] & \text{for } y \notin \mathit{fn}(M) \cup \{y\} & \text{(Struct Res Agent)} \\
 & & \\
x[(L;M);N;P] \equiv x[L;(N;M);P] & & \text{(Struct Meth Assoc)} \\
x[m(\vec{x}) \triangleright P; n(\vec{y}) \triangleright Q; M; R] \equiv x[n(\vec{y}) \triangleright Q; m(\vec{x}) \triangleright P; M; R] & \text{for } n \neq m & \text{(Struct Meth Comm)} \\
x[m(\vec{x}) \triangleright P; m(\vec{y}) \triangleright Q; M; R] \equiv x[m(\vec{y}) \triangleright Q; M; R] & & \text{(Struct Meth Over)} \\
 & & \\
P \equiv P & & \text{(Struct Refl)} \\
P \equiv Q \Rightarrow Q \equiv P & & \text{(Struct Comm)} \\
P \equiv Q, Q \equiv R \Rightarrow P \equiv R & & \text{(Struct Trans)} \\
 & & \\
P \equiv Q \Rightarrow (\boldsymbol{\nu}\, v)P \equiv (\boldsymbol{\nu}\, v)Q & & \text{(Struct Cong New)} \\
P \equiv Q \Rightarrow P \mid R \equiv Q \mid R & & \text{(Struct Cong Par)} \\
P \equiv Q \Rightarrow x[M;P] \equiv x[M;Q] & & \text{(Struct Cong Agent)} \\
P \equiv Q \Rightarrow \alpha . P \equiv \alpha . Q & & \text{(Struct Cong Action)}
\end{array}
$$

Figure 2: Structural Congruence for Agents

The first block of clauses are standard (they are rules of the π-calculus). The rule (Struct Res Agent) modifies the rule for agents in the Ambient and Seal calculi to account for the presence of methods. The (Struct Meth *) clauses are specific of our agents and establish agent equivalence up to reordering of method sequences. The last two blocks are the standard equivalence and congruence rules. In addition, we identify processes up to renaming of bound names: $(\boldsymbol{\nu}\, x)P = (\boldsymbol{\nu}\, y)P\{^{y}/_{x}\}$ if $y \notin \mathit{fn}(P)$. Structural congruence is given parametrically in the definition of actions and fn which varies depending on the specific calculus at issue (see Sections 3 and 4 and the Appendix).

The behavior of agents is now defined in terms of a reduction relation which obeys the structural rules in Figure 3, plus specific rules for each of the construct in the calculus.

[2] *self* may only occur within the agents' methods M, not in the process P

$$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \Rightarrow P' \rightarrow Q' \qquad P \rightarrow Q \Rightarrow x[P] \rightarrow x[Q]$$
$$P \rightarrow Q \Rightarrow (\boldsymbol{\nu}\, x)P \rightarrow (\boldsymbol{\nu}\, x)Q \qquad P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$$

Figure 3: Structural Rules for Reduction

2.2 Messages and Method Invocation

The semantics of method invocation is based on the idea of *self-substitution* distinctive of the Object Calculi of [1]: since agents are named, what gets substituted for the *self* variable is the *name* of the agent rather than the agent itself. Several choices can then be made as to where the method invocation should occur, and as to where the body of the invoked method should be executed. We single out five possible execution modes, but more options are available.

Code on demand: The invocation occurs in a sibling of the receiver and the body of the selected method is activated at the level of the call:

$$y[N; S \mid x \leftarrow m(\vec{y}).P] \mid x[M; m(\vec{z}) \triangleright Q; R] \ \rightarrow\ y[N; S \mid P \mid Q\{^{\vec{y},x}/_{\vec{z},self}\}] \mid x[M; m(\vec{z}) \triangleright Q; R]$$

This situation corresponds to the case of applets in which the client y requires some code to the server x and executes it locally.

Local servlet: The call occurs outside the receiver and the body of the selected method is activated at the level of the call.

$$x \leftarrow m(\vec{y}).P \mid x[M; m(\vec{z}) \triangleright Q;\ R] \ \rightarrow\ P \mid Q\{^{\vec{y},x}/_{\vec{z},self}\} \mid x[M; m(\vec{z}) \triangleright Q;\ R]$$

This case is similar to the previous one with the only difference that the server is local rather than remote.

Remote method invocation: The invocation occurs in a sibling of the receiver (or in the receiver itself) and the body of the selected method is activated in the receiver.

$$y[N; S \mid x \leftarrow m(\vec{y}).P] \mid x[M; m(\vec{z}) \triangleright Q;\ R] \ \rightarrow\ y[N; S \mid P] \mid x[M; m(\vec{z}) \triangleright Q; Q\{^{\vec{y}}/_{\vec{z}}\} \mid R]$$

$$x[M; m(\vec{z}) \triangleright Q;\ S \mid \mathit{self} \leftarrow m(\vec{y}).P] \ \rightarrow\ x[M; m(\vec{z}) \triangleright Q; S \mid P \mid Q\{^{\vec{y}}/_{\vec{z}}\}]$$

This situation corresponds to the case of a RMI, where the invoked method is executed remotely (unless it is a self-invocation:[3]).

External-internal: The call occurs outside the receiver (or in the recever itself) and the body of the selected method is activated inside the receiver:

$$x \leftarrow m(\vec{y}).P \mid x[M; m(\vec{z}) \triangleright Q;\ R] \ \rightarrow\ P \mid x[M; m(\vec{z}) \triangleright Q;\ Q\{^{\vec{y}}/_{\vec{z}}\} \mid R]$$

$$x[M; m(\vec{z}) \triangleright Q;\ S \mid \mathit{self} \leftarrow m(\vec{y}).P] \ \rightarrow\ x[M; m(\vec{z}) \triangleright Q;\ S \mid P \mid Q\{^{\vec{y}}/_{\vec{z}}\}]$$

Internal-external: The method is invoked by a sibling of the receiver (or by the receiver itself) and the body of the selected method is activated outside the receiver:

$$y[N; S \mid x \leftarrow m(\vec{y}).P] \mid x[M; m(\vec{z}) \triangleright Q; R] \ \rightarrow\ y[N; S \mid P] \mid x[M; m(\vec{z}) \triangleright Q; R] \mid Q\{^{\vec{y},x}/_{\vec{z},self}\}$$

$$x[M; m(\vec{z}) \triangleright Q;\ S \mid \mathit{self} \leftarrow m(\vec{y}).P] \ \rightarrow\ x[M; m(\vec{z}) \triangleright Q; S \mid P] \mid Q\{^{\vec{y},x}/_{\vec{z},self}\}$$

[3]Note that in this and in the following mode we do not perform self-substitution. Therefore contrary to what affirmed in Footnote 2, in the internal mode *self* may appear outside methods. We could have performed the *self* substitution and used y instead of *self* in the redex of the second rule, but we preferred this solution because it ensures that agent renaming does not affect the semantics of *self*

In the first two execution modes the selected method is activated at the level of the caller; in the following two modes the selected method is activated inside the receiver; in the last mode the interaction between the caller and the receiver activates the selected method at top level.

Note that the activation at the level of the caller corresponds to a unique reduction rule that handles both regular and self method invocations.

The last two modes have in common that they are hybrid, in the sense that the call and the method execution are not performed at the same level. At a closer look, these hybrid modes do not work smoothly; in both cases method invocations for siblings do not behave well if they occur in a method body: the invocation must be encapsulated in an agent in the internal-external case, and extracted from an agent in the external-internal case.

The other modes, instead, are more natural. The RMI mode appears to be better behaved in terms of security, as method bodies do not escape the protected domain of the receiver. On the other hand RMI requires an ad hoc treatment of self calls, which are instead treated uniformely by the local-servlet or code-on-demand modes (althought they may not have the standard semantics in the presence of homonymous agents). Furthermore the ability distinctive of these modes to escape from the receiver turns out to be useful in several situations; this is expecially true for the local-servlet mode which further allows an intra-level synchronization (see for instance the coding of replication, below). As a consequence and for the sake of simplicity, throughout the paper we employ the local-servlet mode, and establish the following reduction rule for messages.

$$\textbf{(send)} \qquad x \leftarrow m(\vec{z}).P \mid x[M; m(\vec{y}) \triangleright Q; R] \quad \rightarrow \quad P \mid Q\{^{\vec{z},x}/_{\vec{y},self}\} \mid x[M; m(\vec{x}) \triangleright Q; R]$$

Figure 4: Reduction for Messages

2.3 Replication

The behavior of replication in concurrent calculi is typically defined by a structural equivalence rule establishing that $!P \equiv !P \mid P$. With our agents we can encode a similar construct relying upon the implicit form of recursion underlying the reduction of method invocation. We first illustrate the idea with an example and then give a more precise definition. Consider the following configuration:

$$\Omega \triangleq x[\mathtt{bang}() \triangleright \mathit{self} \leftarrow \mathtt{bang}();] \mid x \leftarrow \mathtt{bang}()$$

Given the self-substitution semantics of message send, it is a routine check to verify that $\Omega \rightarrow \Omega$ is the only possible rewriting for Ω. To account for replication for a process P, it is then enough to code the body of `bang` as a parallel composition of a message to self and P. Specifically, defining

$$!P \triangleq (\boldsymbol{\nu}\, x)(x[\mathtt{bang}() \triangleright (\mathit{self} \leftarrow \mathtt{bang}() \mid P);] \mid x \leftarrow \mathtt{bang}()) \qquad \text{for } x \notin \mathit{fn}(P)$$

it is easy to verify that for every process P, the processes $!P$ reduces in one step into $P \mid !P$ as desired.[4]

This concludes the discussion on the general model. In the next two sections we illustrate two instances of the model, by endowing agents with different mobility (and communication) primitives.

3 Ambjects

As we anticipated, ambjects combine the functionalities of Ambients [3, 5] and objects: Processes and methods are defined as in Figure 1, actions are defined below.

[4]Similarly we can encode guarded replication $!\alpha.P$ —where replication is performed only after the consumption of α— as follows: $(\boldsymbol{\nu}\, x)(\alpha.x \leftarrow \mathtt{bang}() \mid x[\mathtt{bang}() \triangleright (\alpha.\mathit{self} \leftarrow \mathtt{bang}() \mid P);])$.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{Actions}} \\
\alpha & ::= & \mathtt{in}\ x & \text{enter into } x \\
 & | & \mathtt{out}\ x & \text{exit out of } x \\
 & | & \mathtt{open}\ x & \text{open } x \\
 & | & x \leftarrow m(\vec{y}) & \text{message to } x
\end{array}
$$

The `in` and `out` actions provide ambjects with the same mobility capabilities as ambients, `open` allows an ambject to break through the boundary of its enclosing ambjects and incorporate their body, and messages allow ambjects to exchange the services implemented by their own methods. Notice that the calculus does not include any construct or resource for communication or syncronization: ambjects only communicate via messages, and message sends are syncronous. As we shall illustrate, a form of asynchronous process communication can be encoded.

3.1 Reduction Semantics

The definition of structural congruence, structural reduction as well as the reduction rule for method invocation are inherited directly from the general model. In Figure 5, we give the reduction rules distinctive of ambjects.

$$
\begin{array}{lrcl}
\textbf{(in)} & y[M; \mathtt{in}\ x.P \mid Q] \mid x[N;R] & \rightarrow & x[N; R \mid y[M; P \mid Q]] \\
\textbf{(out)} & x[M; y[N;\ \mathtt{out}\ x.P \mid Q] \mid R] & \rightarrow & y[N; P \mid Q] \mid x[M;R] \\
\textbf{(open)} & x[M;\ \mathtt{open}\ y.P \mid y[N;Q] \mid R] & \rightarrow & x[M;N; P \mid Q \mid R]
\end{array}
$$

Figure 5: Reduction Rules for Ambjects

The reduction rules for the `in` and `out` actions are defined exactly as they are in the Ambient Calculus. Instead, `open` is defined differently, in that (i) it is only reduced within an ambject, and (ii) it merges the method suites of the opening and opened ambjects. The semantics of merging two method suites is defined by structural equivalence: specifically, if a method name is defined both in M and N, then in $M;N$ the definition found in N overrides the definition of M. This notion of merging is not new: in fact, it is directly inspired by the bookkeeping relation introduced in [10] and related to similar constructs found in the literature on objects (see for instance [9]).

3.2 Example: Updatable Ambjects

The standard operation of method override found in object-based calculi [1, 10] can be coded in the core calculus. The intended behavior of a method update can be formulated, informally, as follows: given the ambject $x[M; m(\vec{y}) \triangleright P; Q]$ we wish to replace the current definition $(\vec{y})P$ of m by the new definition $(\vec{z})P'$ to form the ambject $x[M; m(\vec{z}) \triangleright P'; Q]$. Given the untyped nature of the calculus, in principle updates may not only modify the body of a method, but also the formal parameters of that method. When equipped with types, the type system will ensure that updates behave in a more disciplined manner.

Updates can be coded using a distinguished ambject as "updater". The updater carries the new method body and enters the updatable ambject x, while the updatable ambject is coded as an ambject whose controlling process opens the updater thus allowing updates on its own methods. The coding is defined precisely below. We give two different encodings: in the first we have a form of concurrent update, where updates are processes; in the second, updates are "sequential" and coded as actions.

Updates as Processes. Update processes are denoted by $m@x \Leftarrow (\vec{z})P$, read "the m method at x gets definition P and formal parameters $\vec{z}$". We define their behavior as follows: let first

$$m@x \Leftarrow (\vec{z})P \ \stackrel{\triangle}{=}\ \textsc{upd}[m(\vec{z}) \triangleright P;\ \mathtt{in}\ x].$$

Then define an updatable ambject as follows

$$x^\circ[M;P] \stackrel{\triangle}{=} [M;\ !(\texttt{open}\ \text{UPD}) \mid P].$$

Now, if we form the composition $m@x \Leftarrow (\vec{z})P' \mid x^\circ[M; m(\vec{x}) \triangleright P;\ Q]$, the reduction for open enforces the expected behavior. In fact, using the definitions above, we have:

$$\begin{array}{rcl}
m@x \Leftarrow (\vec{z})P' \mid x^\circ[M; m(\vec{y}) \triangleright P;\ Q] & \rightarrow & x^\circ[M; m(\vec{y}) \triangleright P; Q \mid \text{UPD}[m(\vec{z}) \triangleright P';]] \\
& \equiv & x^\circ[M; m(\vec{y}) \triangleright P; Q \mid \texttt{open}\ \text{UPD} \mid \text{UPD}[m(\vec{z}) \triangleright P';]] \\
& \rightarrow & x^\circ[M; m(\vec{y}) \triangleright P; m(\vec{z}) \triangleright P';\ Q] \\
& \equiv & x^\circ[M; m(\vec{z}) \triangleright P';\ Q]
\end{array}$$

The encoding works well also with "self inflicted" updates. Consider:

$$x \leftarrow m().P \mid x^\circ[m() \triangleright (n@\mathit{self} \Leftarrow (\vec{z})Q'); n(\vec{y}) \triangleright Q; R]$$

This reduces to

$$P \mid x^\circ[m() \triangleright (n@\mathit{self} \Leftarrow (\vec{z})Q'); n(\vec{z}) \triangleright Q'; R]$$

as expected. With an appropriate use of restrictions it is possible to establish update permissions. For example for the ambject $(\boldsymbol{\nu}\,\text{UPD})x[M; P \mid !\texttt{open}\ \text{UPD}]$ allows only self-inflicted updates.

Updates as Actions. Update actions are defined similarly to update processes. Notation-wise, we distinguish the two forms using $:=$ for update actions: the process in place of $\Leftarrow$. $(m@x := (\vec{z})P).Q$ first updates m at x and then continues as Q. This behavior can be accounted for by instrumenting the encoding we just described with a "locking" mechanism that blocks Q until the update is completed. An example of how this locking can be implemented is described below:

$$(m@x := (\vec{z})P).Q \stackrel{\triangle}{=} (\boldsymbol{\nu}\,u)(\text{UPD}[m(\vec{z}) \triangleright P;\ \texttt{in}\ x.u[\texttt{k}() \triangleright Q;\ \texttt{out}\ x]] \mid u \leftarrow \texttt{k}() \triangleright)$$

As for update processes, the updater carries the new method body, and enters the ambject x whose method is to be updated. Now, however, the process Q is "locked" in the ambject u: Q will be activated by the message send $u \leftarrow \texttt{k}()$ that invokes the "continuation" method k locking Q. Sequentiality is enforced by the fact that the continuation method may be effectively invoked only after the update is completed, i.e. only after UPD has been opened by the ambject x, and u can exit x. This intuitive explanation is formally justified below, where we assume $u \notin \mathit{fn}(P \mid Q \mid R)$:

$$\begin{array}{ll}
\multicolumn{2}{l}{(m@x := (\vec{z})P').Q \mid x^\circ[M; m(\vec{y}) \triangleright P;\ R]} \\
\equiv & (\boldsymbol{\nu}\,u)(x^\circ[M; m(\vec{y}) \triangleright P;\ R] \mid \text{UPD}[m(\vec{z}) \triangleright P';\ \texttt{in}\ x.u[\texttt{k}() \triangleright Q;\ \texttt{out}\ x]] \mid u \leftarrow \texttt{k}()) \\
\rightarrow & (\boldsymbol{\nu}\,u)(x^\circ[M; m(\vec{y}) \triangleright P;\ R \mid \text{UPD}[m(\vec{z}) \triangleright P';\ u[\texttt{k}() \triangleright Q;\ \texttt{out}\ x]]] \mid u \leftarrow \texttt{k}()) \\
\rightarrow^* & (\boldsymbol{\nu}\,u)(x^\circ[M; m(\vec{z}) \triangleright P';\ R \mid u[\texttt{k}() \triangleright Q;\ \texttt{out}\ x]] \mid u \leftarrow \texttt{k}()) \\
\rightarrow & (\boldsymbol{\nu}\,u)(x^\circ[M; m(\vec{z}) \triangleright P';\ R] \mid u[\texttt{k}() \triangleright Q;] \mid u \leftarrow \texttt{k}()) \\
\equiv & x^\circ[M; m(\vec{z}) \triangleright P';\ R] \mid (\boldsymbol{\nu}\,u)(u[\texttt{k}() \triangleright Q;] \mid u \leftarrow \texttt{k}()) \\
\rightarrow & x^\circ[M; m(\vec{z}) \triangleright P';\ R] \mid Q \mid (\boldsymbol{\nu}\,u)u[\texttt{k}() \triangleright Q;] \\
\approx & x^\circ[M; m(\vec{z}) \triangleright P';\ R] \mid Q
\end{array}$$

The last step in the previous reasoning relies on the bisimilarity $(\boldsymbol{\nu}\,x)x[B] \approx \mathbf{0}$, which holds in the Ambient Calculus if $x \notin \mathit{fn}(B)$ [5]. Although we haven't discussed bisimilarity yet, we may already argue informally that the same relation holds for our ambjects. Intuitively, the relation $(\boldsymbol{\nu}\,x)x[B] \approx \mathbf{0}$ holds for ambients because all the interactions among ambients are based on the use of names. The same is true of ambjects: specifically, the only additional interactions of an ambject $x[B]$ with its surrounding context result from message sends invoking one of x's methods. Since message sends must name the recipient ambject, and $x \notin \mathit{fn}(B)$ ensures that the ambject cannot rename itself, then no interaction is available between $(\boldsymbol{\nu}\,x)x[B]$ and its surrounding context: hence $(\boldsymbol{\nu}\,x)x[B] \approx \mathbf{0}$ whenever $x \notin \mathit{fn}(B)$ as desired.

3.3 Example: Extensible Ambjects

Object calculi often support constructs for extending the structure of objects with the addition of new methods. A corresponding construct can be provided for ambjects: given the ambject $x[M; Q]$ we wish to extend M with a new method $m(\vec{y}) \triangleright P$ to form the ambject $x[M; m(\vec{y}) \triangleright P; Q]$. Method addition may clearly be accomplished by means of exactly same technique we illustrated in the previous section. As in the case of updates, method addition may be coded as a process or an action.

3.4 Example: Communication

A final example shows that communication between processes can also be encoded. As anticipated, we give an example of asynchronous communication. A similar model of channel-based communication is presented in [5], based on the more primitive form of local and *anonymous* communication defined for the Ambient Calculus: here, instead, we rely on the ability to exchange values between methods distinctive of our ambjects.

Channels are modeled as particular ambjects. An "ambject channel" n is an updatable ambject with two methods: a dispatcher method `d` which is invoked by a process wishing to write on n, and a continuation method `k` that is invoked by the dispatcher when a process is ready to read from n.

$$
\begin{array}{rcl}
(ch\ n) & \triangleq & n^{\circ}[\mathtt{d}(x) \triangleright \mathit{self} \leftarrow \mathtt{k}(x); \mathtt{k}(x) \triangleright \mathit{self} \leftarrow \mathtt{d}(x);] \\
\overline{n}(y).Q & \triangleq & n \leftarrow \mathtt{d}(y).Q \\
n(x).P & \triangleq & (\mathtt{k}@n := (x)(\mathtt{k}@\mathit{self} := (x)\mathit{self} \leftarrow \mathtt{d}(x)).P)
\end{array}
$$

The communication protocol is as follows. The process reading from $(ch\ n)$ installs its continuation P as the body of the `k` method of n. The process writing y on n sends the message $\mathtt{d}(y)$ to $(ch\ n)$. At this stage, the `d` method in n invokes `k` with the value it received from Q thus activating P. Finally, P restores the original definition of `k` in n and then continues. This informal reasoning is justified below, where $reset_{\mathtt{k}}(m)$ is short for $\mathtt{k}@m := (x)m \leftarrow \mathtt{d}(x)$

$$
\begin{array}{rcl}
(ch\ n) \mid n(x).P \mid \overline{n}(y).Q & \rightarrow & n^{\circ}[\mathtt{d}(x) \triangleright \mathit{self} \leftarrow \mathtt{k}(x); \mathtt{k}(x) \triangleright reset_{\mathtt{k}}(\mathit{self}).P;] \mid \overline{n}(y).Q \\
& \rightarrow & n^{\circ}[\mathtt{d}(x) \triangleright \mathit{self} \leftarrow \mathtt{k}(x); \mathtt{k}(x) \triangleright reset_{\mathtt{k}}(\mathit{self}).P;] \mid Q \mid n \leftarrow \mathtt{k}(y) \\
& \rightarrow & n^{\circ}[\mathtt{d}(x) \triangleright \mathit{self} \leftarrow \mathtt{k}(x); \mathtt{k}(x) \triangleright reset_{\mathtt{k}}(\mathit{self}).P;] \mid Q \mid reset_{\mathtt{k}}(n).P\{^{y}/_{x}\} \\
& \rightarrow & (ch\ n) \mid P\{^{y}/_{x}\} \mid Q
\end{array}
$$

The initial (circular) definitions of the two methods inside n ensures that values written on n do not get lost even though there is no process willing to read from n. A more rigorous coding of communication should also include a locking mechanism to prevent interferences to the protocol.

4 Sealjects

Unlike the Ambient Calculus, where mobility is accounted for by actions, in Seal mobility is obtained by communicating agents (in fact, agent bodies) on channels. Also, channels are located resources, in that they belong to a particular agent. Therefore, synchronization between two agents requires that the two partners specify not only the channel, but also the channel's location. For instance, $\overline{x}^{n}(y)$ writes y on the channel x located in the seal n, while $z^{\uparrow}\{\!|x|\!\}$ waits to receive the body of a seal on the channel z, located in its surrounding context (denoted by $\uparrow$): once the body is received it is is framed in a new seal named x (see the reduction rules below).

In defining *sealjects*, we use a simplified version of the Seal Calculus, where we disregard portals and their operations, and we restrict to a *linear* version of the calculus described in [19] where the receive action is monadic. Processes and methods are defined as if Figure 1, actions are defined below.

Actions			
α	$::=$	$x^{\eta}(\vec{x})$	input
	$\mid$	$\overline{x}^{\eta}(\vec{x})$	output
	$\mid$	$\overline{x}^{\eta}\{x\}$	send
	$\mid$	$x^{\eta}\{x\}$	receive
	$\mid$	$x \leftarrow m(\vec{x})$	message send

Locations			
η	$::=$	x	down
	$\mid$	$*$	local
	$\mid$	$\uparrow$	up

4.1 Reduction Semantics

Structural congruence, the rules for structural reduction as well as the reduction rule for method invocation are defined exactly as in the general model. The definition of the semantics of sealjects is completed by the reduction rules in Figure 6.

(local)	$x^{*}(\vec{u}).P \mid \overline{x}^{*}(\vec{v}).Q$	$\rightarrow$	$P\{^{\vec{v}}/_{\vec{u}}\} \mid Q$
(write out)	$x^{\eta_1}(\vec{u}).P \mid y[M;\overline{x}^{\eta_2}(\vec{v}).Q\|R]$	$\rightarrow$	$P\{^{\vec{v}}/_{\vec{u}}\} \mid y[M;Q\|R]$
(write in)	$\overline{x}^{\eta_1}(\vec{u}).P \mid y[M;x^{\eta_2}(\vec{v}).Q\|R]$	$\rightarrow$	$P \mid y[M;Q\{^{\vec{u}}/_{\vec{v}}\}\|R]$
(move out)	$x^{\eta_1}\{u\}.P \mid y[M;\overline{x}^{\eta_2}\{v\}.Q\|v[B]\|R]$	$\rightarrow$	$P \mid u[B] \mid y[M;Q\|R]$
(move in)	$\overline{x}^{\eta_1}\{u\}.P \mid u[B] \mid y[M;x^{\eta_2}\{v\}.Q\|R]$	$\rightarrow$	$P \mid y[M;Q\|R\|v[B]]$

where either $\eta_1 = *$ and $\eta_2 =\uparrow$, or $\eta_1 = y$ and $\eta_2 = *$.

Figure 6: Reduction Rules for Sealjects

Note that, contrary to what happens in Ambients (and Ambjects), in Seal(jects) mobility of an agent is the result of a mutual agreement between a sending environment and a receiving environment, and the agent has no role in it (the use of $\uparrow$ ensure that the agent will be dynamically bounded to the new enclosing environment).

4.2 Example: Mini Java

We illustrate the use of Sealjects in the specification of the subset of Java described below. We use c to range over class names, x over variables, and m over messages and method names. Types, denoted by T are either the type `void` (the type of statements), or a class name. Finally, modifiers, denoted by V are either `private` or `public`. The rest of the syntax is specified by the following productions.

Class Sequences

$C \ ::= \texttt{class}\ c\ \texttt{\{}A\ M\texttt{\}} \ \mid\ C\texttt{;}C$

Attribute Sequences

$A \ ::= V\ T\ x\texttt{;} \ \mid\ V\ T\ x\ \texttt{=}\ E\texttt{;} \ \mid\ A\ A$

Method Sequences

$M \ ::= V\ T\ m\texttt{(}T_1 x_1\texttt{,}\ldots\texttt{,}T_n x_n\texttt{)\{}B\texttt{\}} \ \mid\ M\texttt{;}M \quad (n \geq 0)$

Bodies

$B \ ::= S \ \mid\ S\texttt{;return(}E\texttt{)}$

Statements

S	$::= x\ \texttt{=}\ E$	assignment to the *global* variable x
	$\mid\ E\texttt{.}x\ \texttt{=}\ E$	assignment to the variable x of E
	$\mid\ S\texttt{;}S$	sequencing
	$\mid\ E\texttt{.}m(E_1\texttt{,}\ldots\texttt{,}E_n)$	method call $(n \geq 0)$

Object Expressions

E	$::= x$	global variable
	$\mid E.x$	attribute of E
	$\mid$ `null`	undefined object
	$\mid$ `this`	*self*
	$\mid$ `new` c	a new instance of the class c
	$\mid E.m(E_1, \ldots, E_n)$	method call ($n \geq 0$)

As usual, a program is a non-empty sequence of classes. One (and only one) of these classes declares the method `main`: this class, we denote by $\hat{c}$, cannot be instantiated and its public fields are the global variables of the program. Mini Java appears to be a realistic representative for Java, with the only noticeable exception that methods do not have local variables. Local variables are not included here as they would require a cumbersome encoding (they should be stored outside the seal).

From Mini Java to Sealject The specification of Mini Java with Sealjects is given in the form of a translation. The model of classes that results from this translation is *object-based* and is reminiscent of the *record-of-premethods* model of [1]: a class is interpreted as a sealject with a unique, distinguished, method *new* which receives as argument the object identifier (oid) of the instance to be created. When invoked, *new* packages the class methods and attributes into a new sealject and returns it. The definition is as follows[5]

$$[\![\texttt{class } c\{A_1;\ldots;A_m;M_1;\ldots;M_n\}]\!] \\ = c[new(o) \triangleright (\boldsymbol{\nu}\, r\_x_1, w\_x_1, \ldots, r\_x_h, w\_x_h, m_1, \ldots, m_k)\, o[[\![M_1]\!];\ldots;[\![M_n]\!];[\![A_1]\!] \mid \ldots \mid [\![A_m]\!]];]$$

where $x_1, \ldots, x_h$ and $m_1, \ldots, m_k$ respectively are the private attributes and methods of c.

A different interpretation must be given for the distinguished class $\hat{c}$ with the method `main()`, since it cannot be instantiated, and it provides the global variables:

$$[\![\texttt{class } \hat{c}\{A_1;\ldots;A_m;M_1;\ldots;M_n\}]\!] \\ = (\boldsymbol{\nu}\, r\_x_1, w\_x_1, \ldots, r\_x_h, w\_x_h, m_1, \ldots, m_k)\hat{c}[[\![M_1]\!];\ldots;[\![M_n]\!];[\![A_1]\!] \mid \ldots \mid [\![A_m]\!]]$$

A program, that is a sequence of single class declarations is interpreted as the parallel composition below.

$$[\![C_1;C_2;\ldots;C_n]\!] = [\![C_1]\!] \mid [\![C_2]\!] \mid \ldots \mid [\![C_n]\!] \mid \hat{c} \leftarrow main()$$

The rest of the specification is given in terms of four translation mappings, one for each syntactic category in Mini Java. Each global variable (or attribute) x is associated with a corresponding (private) channel holding the variable's current value, and two channels, r_x and w_x which are used to read and write variable values. The read and write channels for global variables are located in the sealject corresponding to the distinguished class $\hat{c}$.

The translation of expressions depends on a channel ℓ, which is used to output the value resulting from evaluating the expression. In the interpretation of `new`, a new oid o is created for the class instance, passed along to with the call to *new*, and then communicated via ℓ to users of the instance: as a result, oids cannot be manipulated or generated by users. In the translation of method invocation, the channel r is used to receive the results of evaluating the expressions: these results are then recorded in o_i. An alternative definition could be given using different r_i's to evaluate the expressions in parallel.
The interpretation of statements, in Figure 8, also depends on a channel ℓ: in this case, however, the channel is only used for synchronization. The translation is completed by the definitions for attributes and methods, in Figure 9. The definition is given regardless of the modifiers, as modifiers are taken care of the interpretation of classes, using the restriction operator.

[5]An alternative object based model, where classes are viewdas prototypes, and class instances are created by cloning can be given in the non-linear version of the calculus [19] relying on the duplication primitives available for for sealjects.

$$
\begin{array}{lcl}
\llbracket x \rrbracket(\ell) & = & \overline{r\_x}^{\hat{c}}(\ell) \\
\llbracket E\,.\,x \rrbracket(\ell) & = & (\boldsymbol{\nu}\,\ell_1)(\llbracket E \rrbracket(\ell_1) \mid \ell_1(y)\,.\,\overline{r\_x}^{y}(\ell)) \\
\llbracket \texttt{null} \rrbracket(\ell) & = & \overline{\ell}(\mathit{null}) \\
\llbracket \texttt{this} \rrbracket(\ell) & = & \overline{\ell}(\mathit{self}) \\
\llbracket \texttt{new}\ c \rrbracket(\ell) & = & (\boldsymbol{\nu}\,o)(c \leftarrow \mathit{new}(o)\,.\,\overline{\ell}(o)) \\
\llbracket E\,.\,m\texttt{(}E_1\texttt{,}E_2\texttt{,}\,\ldots\,\texttt{,}E_n\texttt{)} \rrbracket(\ell) & & \\
 & = & (\boldsymbol{\nu}\,r)(\llbracket E \rrbracket(r) \mid r(o)\,. \\
 & & \quad (\llbracket E_1 \rrbracket(r) \mid r(o_1)\,. \\
 & & \quad (\llbracket E_2 \rrbracket(r) \mid r(o_2)\,. \\
 & & \qquad \vdots \\
 & & \quad (\llbracket E_n \rrbracket(r) \mid r(o_n)\,. \\
 & & \quad o \leftarrow m(o_1, o_2 \ldots, o_n, \ell)\ )\ldots))) \\
\llbracket S\texttt{;return(}E\texttt{)} \rrbracket(\ell) & = & (\boldsymbol{\nu}\,l_1)(\llbracket S \rrbracket(\ell_1) \mid \ell_1()\,.\,\llbracket E \rrbracket(\ell))
\end{array}
$$

Figure 7: $\llbracket\cdot\rrbracket$: Expressions $\rightarrow$ Channels $\rightarrow$ Processes

$$
\begin{array}{lcl}
\llbracket x\ \texttt{=}\ E \rrbracket(\ell) & = & (\boldsymbol{\nu}\,\ell_1)(\llbracket E \rrbracket(\ell_1) \mid \ell_1(v)\,.\,\overline{w\_x}^{\hat{c}}(v,\ell))) \\
\llbracket E_1\,.\,x\ \texttt{=}\ E_2 \rrbracket(\ell) & = & (\boldsymbol{\nu}\,\ell_1)(\llbracket E_1 \rrbracket(\ell_1) \mid \ell_1(o)\,.\,(\llbracket E_2 \rrbracket(\ell_1) \mid \ell_1(v)\,.\,\overline{w\_x}^{o}(v,\ell))) \\
\llbracket S_1\texttt{;}S_2 \rrbracket(\ell) & = & (\boldsymbol{\nu}\,\ell_1)(\llbracket S_1 \rrbracket(\ell_1) \mid \ell_1()\,.\,\llbracket S_2 \rrbracket(\ell)) \\
\llbracket E\,.\,m\texttt{(}E_1\texttt{,}E_2\texttt{,}\,\ldots\,\texttt{,}E_n\texttt{)} \rrbracket(\ell) & & \\
 & = & (\boldsymbol{\nu}\,r)(\llbracket E \rrbracket(r) \mid r(o)\,. \\
 & & \quad (\llbracket E_1 \rrbracket(r) \mid r(o_1)\,. \\
 & & \quad (\llbracket E_2 \rrbracket(r) \mid r(o_2)\,. \\
 & & \qquad \vdots \\
 & & \quad (\llbracket E_n \rrbracket(r) \mid r(o_n)\,. \\
 & & \quad o \leftarrow m(o_1, o_2 \ldots, o_n, \ell)\ )\ldots)))
\end{array}
$$

Figure 8: $\llbracket\cdot\rrbracket$: Statements $\rightarrow$ Channels $\rightarrow$ Processes

$$
\begin{array}{lcl}
\llbracket V\ T\ x\texttt{;} \rrbracket & = & \llbracket V\ T\ x\ \texttt{= null;} \rrbracket \\
\llbracket V\ T\ x\ \texttt{=}\ E\texttt{;} \rrbracket & = & (\boldsymbol{\nu}\,\mathit{cell})(\llbracket E \rrbracket(\mathit{cell}) \\
 & & \quad \mid\ !(w\_x(v_{new},\ell)\,.\,\mathit{cell}(v_{old}).\overline{\ell}^{\uparrow}()\,.\,\overline{\mathit{cell}}(v_{new})) \\
 & & \quad \mid\ !(r\_x(\ell)\,.\,\mathit{cell}(v).\overline{\ell}^{\uparrow}(v)\,.\,\overline{\mathit{cell}}(v)) \\
\llbracket V\ T\ m\texttt{(}T_1x_1\texttt{,}\,\ldots\,\texttt{,}T_nx_n\texttt{)\{}B\texttt{\}} \rrbracket & = & m(x_1,\ldots,x_n,\ell) \triangleright \llbracket B \rrbracket(\ell)
\end{array}
$$

Figure 9: $\llbracket\cdot\rrbracket$: Attributes $\rightarrow$ Processes, $\qquad \llbracket\cdot\rrbracket$: Methods $\rightarrow$ Processes

4.3 Example: ProActive

Although the specification of MiniJava uses parallel composition of processes, the resulting semantics is, in fact, strictly sequential: in particular, a statement is activated only after the previous statement is finished. Adding true concurrency to the language is quite straighforward: just add to the statements of MiniJava $S \mid S$ and interpret it as $[\![S_1 \mid S_2]\!](\ell) = (\boldsymbol{\nu}\, \ell_1, \ell_2)([\![S_1]\!](\ell_1) \mid [\![S_2]\!](\ell_2) \mid \ell_1().\ell_2().\overline{\ell}())$. More challenging, and interesting, is extending the languages with primitives that use mobility. In order to show mobility at work we describe how to specify a distributed programming environment, and we choose ProActive [7], a Java library with transparent active objects and call-by-need remote method invocation (RMI).

ProActive allows one to program a distributed application with minimal effort. When creating an object, the programmer has only to specify the node of the network the new object must be allocated on: the system handles remote method invocations transparently. From a linguistic point of view the difference is minimal: it suffices to add to expressions the following production:

$E ::= \texttt{newActive}\ c(\mathit{node})$

This expression creates a new active object of class c, places it on the node *node* of the distributed system and returns its oid as result. From the programmer's viewpoint, active objects are transparent, since local or remote dispatching of messages is provided by the underlying system, depending on whether receivers are remote or not. The only observable difference is that call-by-need is used for remote method invocations: that is, a thread does not stop its execution to wait for the completion of the call of a remote method; it stops only when it needs the result of the call.

A support for this form of distributed programming can be implemented with Sealjects by extending the interpretation of MiniJava with the following three ingredients:

1. Nodes and network transportation. Objects are no longer implemented by sealjects running at top level. Instead, objects are distributed over some stationary sealjects representing the nodes of the distributed system and transported between these nodes.

2. *Stubs*. They are local representations of remote objects: a stub receives the calls to a given remote object, it forwards them to that object, and handles possible results.

3. *Futures*. They are objects used to implement call-by-need. Intuitively, a future is "virtual" representation of the result of a remote method invocation: any call to the result of a RMI is sent to the future associated to the result, and the future forwards the call to the actual result (or its stub), as soon as the latter is available (this is similar to the *polling* asynchronous request invocation in the next version of CORBA [18]).

In order to simplify the translation we assume that all instance variables of a class are private (as customary a public variable `x` can be simulated by a pair of `get_x` and `set_x` methods.)

In Figure 4.3 we show an example of ProActive's execution where a remote object is created and a message sent to it.

The distributed system consists of two sealjects Node1 and Node2 and a network process that handles mobility and communications between nodes. Part A of the figure represents a initial state in which an object $o1$, denoted by the variable x, is present on Node1 (this may correspond to the configuration resulting from executing `x:= new c` for some class `c`). Part B shows what happens when, say, $o1$ executes `y:=newActive c(Node2)`: a new active object $o2$ and its stub are created on Node1 and $o2$ is moved to Node2 through the network. At this stage, if $o1$ sends a message `m()` to the newly created object, as in `y.m()`, the message is handled by the stub, which forwards the message to $o2$ on Node2. When the method returns a result, the stub returns to the caller the oid of a future, *fut*, and communicates *fut* to the callee (part C). In part D, $o2$ communicates to the future the result of the call, say, $o1$, and the future redirects to it all the calls.

In order capture this behavior, we modify the translation of classes so that, besides providing the *new* method for creating new instances, the class also provides a *new_stub* method for creating stubs for

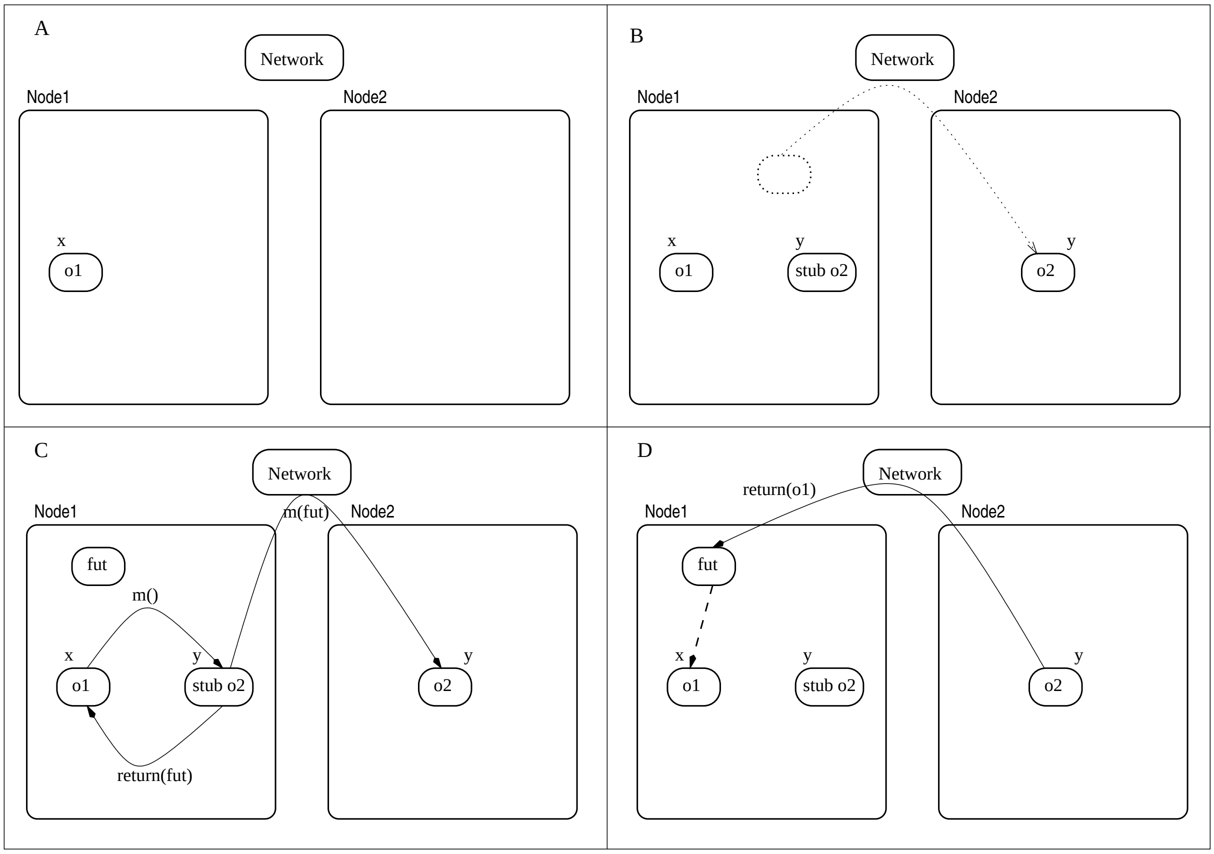

Figure 10: Execution in ProActive

the class' instances (in case one of the instances is executed remotely), as well as a *new_fut* method for creating new futures (in case one of the instances is the result of a RMI):

$$
\begin{aligned}
&[\![\texttt{class}\ c\texttt{\{}A_1\,\texttt{;}\ldots\texttt{;}A_m\,\texttt{;}M_1\,\texttt{;}\ldots\texttt{;}M_n\texttt{\}}]\!] = \\
&\qquad c[\mathit{new}(o) \triangleright (\boldsymbol{\nu}\, r\_x_1, w\_x_1, \ldots, r\_x_n, w\_x_n, m_1, \ldots, m_k)\, o[[\![M_1]\!]; \ldots; [\![M_n]\!]; [\![A_1]\!] \mid \ldots \mid [\![A_m]\!]]; \\
&\qquad\quad \mathit{new\_stub}(o,n) \triangleright (o[\mathcal{S}[\![M_{k+1}]\!](n); \ldots; \mathcal{S}[\![M_n]\!](n);]); \\
&\qquad\quad \mathit{new\_fut}(f,r) \triangleright (r^{\uparrow}(o)\,.\,f[\mathcal{F}[\![M_{k+1}]\!](o); \ldots; \mathcal{F}[\![M_n]\!](o);]);\ ]
\end{aligned}
$$

Again, $m_1, \ldots, m_k$ are the private methods of c, and $M_{k+1}, \ldots, M_n$ are the public method fields of c. The *new_stub* method is parametrized by the oid o and the node n of the active object the stub must represent and the *new_fut* method is parametrized by the oid f of the future and the channel r on which the result object is returned. The primitive `newActive` may now be translated as follows:

$$[\![\texttt{newActive}\ c(n)]\!](\ell) \quad = \quad (\boldsymbol{\nu}\, o)(c \leftarrow \mathit{new}(o)\,.\,(\boldsymbol{\nu}\, x)(\overline{\mathit{move}}^{\uparrow}(x,o,n)\,.\,\overline{x}^{\uparrow}\{o\}\,.\,c \leftarrow \mathit{new\_stub}(o,n)\,.\,\overline{\ell}(o)))$$

When invoked, `newActive` $c(n)$ first creates a new oid o and then obtains an instance of the class c with oid o by sending *new*(o) to the class c; after that, it uses the the public channel *move* to announce to the network that it will send on a private channel x the object o to be moved to the node n; finally, o is sent onto x, a stub for o is created, and the oid o of the new instance returned.

The definition of the methods for futures and stubs is as follows:

$$
\begin{aligned}
\mathcal{F}[\![\texttt{public}\ T\ m\texttt{(}T_1 x_1\texttt{,}\ \ldots\ \texttt{,}T_k x_k\texttt{)\{}B\texttt{\}}]\!](o) &= m(x_1, \ldots, x_k, \ell)(o \leftarrow m(x_1, \ldots, x_k, \ell)) \\
\mathcal{S}[\![\texttt{public}\ c\ m\texttt{(}T_1 x_1\texttt{,}\ \ldots\ \texttt{,}T_k x_k\texttt{)\{}B\texttt{\}}]\!](n) &= (\boldsymbol{\nu}\, f, r)(c \leftarrow \mathit{new\_fut}(f,r)\,.\,\overline{\mathit{rmi\_k}}^{\uparrow}(m, x_1, \ldots, x_k, r, n)\,.\,\overline{\ell}(f)) \\
\mathcal{S}[\![\texttt{public void}\ m\texttt{(}T_1 x_1\texttt{,}\ \ldots\ \texttt{,}T_k x_k\texttt{)\{}B\texttt{\}}]\!](n) &= (\boldsymbol{\nu}\, r)(\overline{\mathit{rmi\_k}}^{\uparrow}(m, x_1, \ldots, x_k, r, n)\,.\,(\overline{\ell}() \mid r^{\uparrow}()))
\end{aligned}
$$

Note the definition of $\mathcal{S}[\![]\!]$: when the method returns a result of class c the stub creates a future of this class, sends to the network the arguments of the RMI (on a public channel rmi_k that depends on the arity k of the method) and the future's oid, and returns the identifier of the future as result. If the return type of the method is `void`, then the stub performs the RMI, acknowledges termination on channel ℓ, and waits to consume the termination acknowledgment from the RMI.

The specification can be completed by defining the processes that implement the network. We only outline the specification here and only mention that, besides object transportation, the network must implement the discipline to handle local arguments of RMIs, by creating remote stubs or migrating part of them, and also garbage collect used futures.

5 Conclusion

One of the main purposes, as well as of the challenges, for a foundational formalism for distributed systems is to establish an adequate setting where formal proofs of behavioral properties for processes and agents can be carried out.

Viewed from this perspective, the work on Ambjects and Sealjects we have described in this draft should be understood as a very preliminary attempt to define a computation model for distributed applications, where conventional technology –based on remote exchange of messages between static sites– and mobile agents coexist and can be integrated in a uniform way. This attempt appears to be well motivated by the current –rather intense– debate on the role of mobility in wide-area distributed applications; a debate in which even proponents and developers of mobile agents offer that "we probably shouldn't expect purely mobile applications to replace other structuring techniques"[12].

Clearly, however, we are far from being in a position where this model can be proposed as a foundational formalism: a lot of work needs to be done in that direction, both to evaluate its adequacy as a formal tool for modelling realistic applications, and to develop a reasonable algebraic theory for the calculus. Both these aspects are current topic of research for the two calculi – Ambients and Seals – on top of which two instances of our general model for mobile objects are defined. The recent papers on typed formulations of Ambients [6, 4, 14] provide rather interesting and useful insight into how an algebraic theory for mobile objects could be defined as well as into the rôle of types in proving behavioral properties.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, 1996.

[2] M. Bugliesi and S. Crafa. Object calculi with dynamic messages. In *FOOL'6 Proc. of the 6th Int. Workshop on Foundations of Object Oriented Languages.* 1999. Electronic Proceedings.

[3] L. Cardelli. Abstractions for mobile computations. In *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science, pages 51–94. Springer, 1999.

[4] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, number 1644 in Lecture Notes in Computer Science, pages 230–239. Springer, 1999.

[5] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL'98.* ACM Press, 1998.

[6] L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM Press, 1999.

[7] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[8] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[9] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proc. of ACM Symp. POPL*, pages 125–135. ACM Press, 1990.

[10] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[11] A. Gordon and P. D Hankin. A concurrent object calculus: reduction and typing. *In Proceedings HLCL'98, Elsevier ENTC*, 1998. Also Technical Report 457, University of Cambridge Computer Laboratory, February 1999.

[12] D. Johansen. Quote from "trend wars". *IEEE Concurrency*, 7(3), Sept 1999.

[13] J. Kleist and Sangiorgi D. Imperative objects and mobile processes. Unpublished manuscript.

[14] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'2000*. ACM Press, 2000. To Appear.

[15] S. Nishimura. Static typing for dynamc messages. In *Proc. of POPL'98*. ACM Press, 1998.

[16] B.C. Pierce and D.N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming, Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, April 1995.

[17] V.T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *ECOOP '94*, number 821 in Lecture Notes in Computer Science, pages 100–117. Springer, 1994.

[18] S. Vinoski. New features for CORBA 3.0. *Communications of the ACM*, 41(10), 1998.

[19] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science. Springer, 1999.

[20] D.J Walker. Objects in the π calculus. *Information and Computation*, 116(2):253–271, 1995.