In CONCUR 2000, Lecture Notes in Computer Science 1877:504-520, Springer, 2000.

Typed Mobile Objects*

Michele Bugliesi[1], Giuseppe Castagna[2], and Silvia Crafa[1]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia
E-mail: {michele,silvia}@dsi.unive.it, Web: http://www.dsi.unive.it/~{michele,silvia}

[2] C.N.R.S., Département d'Informatique, École Normale Supérieure, Paris
E-mail: Giuseppe.Castagna@ens.fr, Web: http://www.di.ens.fr/~castagna.

Abstract. We describe a general model for embedding object-oriented constructs into calculi of mobile agents. The model results from extending agents with methods and primitives for message passing. We then study an instance of the model based on Cardelli and Gordon's Mobile Ambients. We define a type system for the resulting calculus, give a subject reduction theorem, and discuss the rôle of the type system for static detection of run-time type errors and for more program verification purposes.

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.

Drawing on our earlier preliminary work on the subject [2], in this paper we develop a general model for integrating object-oriented constructs into calculi of mobile agents. The resulting framework can be looked at in different ways: (i) as a *concurrent* object calculus where objects have explicit names, in the style of [9], (ii) as a *distributed* object calculus where objects are stored at different named locations, or (iii), more ambitiously, as a model for distributed computation, where conventional client-server technology based on (remote) exchange of message between agents, and mobile agents coexist.

The model results from extending the structure of *named* agents with method definitions and primitives for dealing with message passing and self denotations. The extension has rather interesting payoffs, as it leads to a principled approach to structuring agents. In particular, introducing methods and message passing as primitives, rather than encoding them on top of the underlying calculus of agents leads to a rich and precise notion of agent interface and type. Furthermore, it opens the way to reusing the advances in type system of object-oriented programming and static analysis.

After giving an outline of the general model, we illustrate these aspects by discussing an instance of that model, named MA^{++}, which is based on the calculus of

* Work partially supported by MURST Project 9901403824_003 and by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet".

Mobile Ambients (henceforth MA) of [3, 5]. We illustrate the calculus with a number of rather variated examples: specifically, we show that it is possible to encode primitives like method overriding distinctive of object calculi; we also show that various forms of process communication can be encoded (here, and throughout the paper, we use the terms "encode" and "encoding" in somewhat loose sense: we should in fact use "simulate" and "simulation" as we don't claim these encodings to be "atomic" —i.e. free of interferences— in all possible contexts.).

Then we study the type theory of our calculus. The motivation for doing that is twofold: (i) a sound type system can statically detect run-time type errors such as the classical "message not understood" errors distinctive of object-oriented languages and calculi; (ii) an expressive type system eases the definition of program equivalences and their proof (see for example [13]). The present paper gives an in-depth account of the former aspect, and only hints at the latter (see Section 6), leaving a detailed treatment for future work.

Several proposals of formalisms for foundational study of concurrent object-oriented programming can be found in the literature (e.g., [9, 12, 15, 16, 18, 7, 14]): however, to our knowledge, no previous work directly embeds methods into calculi of mobile agents.

2 Outline of the model

The definition of the model is given parametrically on the underlying calculus of processes and agents so that it can be adapted to formalisms such as Mobile Ambients [5], Safe Ambients [13], or the Seal Calculus [17]. The minimal requirement we assume for the underlying calculus of mobile agents is the existence of the following constructs: **0** denoting the inactive process, $P \mid Q$, denoting the parallel composition of two processes P and Q, $a[P]$, denoting the process (or agent) named a running the process P, $(\nu x)P$, that restricts the name x to P, and finally $A.P$, that performs the action described by the expression A and then continues as P. Clearly, the actions will eventually include 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 locations: furthermore, since naming allows nesting, locations may be structured hierarchically [3]. As a consequence, named processes model both the nodes of the distributed system and the agents over that system.

2.1 Syntax

The generic model of mobile objects results from generalizing the structure of named agents to include method definitions, as in $a[M\,;P]$, where P is a process and M a set (rather, a list) of method definitions. The syntax of agents is defined by the productions in Figure 1.

Notation. In the following we use P, Q, R to range over processes, L, M, N to range over (possibly empty) method sequences, and lower case letters to range over generic

Methods

$M ::=$	$m(\vec{x}_n) \triangleright \varsigma(z)P$	method
$\mid$	M, M	method sequence
$\mid$	ε	empty sequence

Expressions

$A ::=$	$a, b, \ldots, x, y \ldots$	names
$\mid$	$a \texttt{ send } m(\vec{A})$	message
$\mid$	$A.A$	path
$\mid$	ε	empty path

Processes

$P ::=$	$\mathbf{0}$	inactivity
$\mid$	$P \mid P$	parallel composition
$\mid$	$a[M \,;\, P]$	ambject or sealject
$\mid$	$(\nu x)P$	restriction
$\mid$	$A.P$	action

Fig. 1. Syntax of Agents

names, preferring when possible $a, b, \ldots$ for agent names, and $x, y, \ldots$ for (method) parameters. Method names, denoted by m and n range over a disjoint alphabet and have a different status: they are fixed labels that may not be restricted, abstracted, substituted, nor passed as values (they are similar to field labels in record-based calculi). We omit trailing or isolated $\mathbf{0}$ processes and empty method sequences, using A, $a[M\,;\,]$, $a[P]$, and $a[\,]$ to abbreviate $A.\mathbf{0}$, $a[M\,;\,\mathbf{0}]$, $a[\varepsilon\,;\,P]$, and $a[\varepsilon\,;\,\mathbf{0}]$ respectively.

We sometimes write $a[(m_i(\vec{x}_{k_i}) \triangleright \varsigma(z_i)P_i)_{i\in[1..n]}\,;\,P]$ as a shorthand for the agent $a[m_1(\vec{x}_{k_1}) \triangleright \varsigma(z_1)P_1, \ldots, m_n(\vec{x}_{k_n}) \triangleright \varsigma(z_n)P_n\,;\,P]$. Similarly we write $\vec{x}_n$ as a shorthand for $x_1, \ldots, x_n$ and omit the subscript n when there is no riskof ambiguity. Finally, $P\{\vec{x}_n := \vec{y}_n\}$ denotes the term resulting from substituting every free occurrence of x_i by y_i in P; equivalent notations that we also use are $P\{\vec{x}_{n-1} := \vec{y}_{n-1}, x_n := y_n\}$ and $P\{\vec{x}_{n-1}, x_n := \vec{y}_{n-1}, y_n\}$.

Methods. A method definition has the form $m(\vec{x}_n) \triangleright \varsigma(z)P$, where the m is the name of the method, $\vec{x}_n$ the list of its formal parameters, and $\varsigma(z)P$ its body. As in the Object Calculi of [1], the ς-bound variable z is the "internal" name of the host agent: the scope of the ς-binder is the method body P. 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 of methods is abbreviated to $m(\vec{x}_n) \triangleright P$ when z does not occur free in P, or when the presence of the binder is irrelevant to the context in question.

Expressions. The definition and associated behavior of expressions is what distinguishes different calculi for mobility. For example the expressions that perform mobility in the Seal Calculus use channels synchronization, and act on agents that are passive with respect to mobility. In Mobile Ambients, instead, mobility expressions are exercised by the moving agents themselves, without intervention by the surrounding environment (Safe Ambients add a synchronization mechanism to Mobile Ambients). Since we want our extension to be independent of the underlying primitives for mobility, at the present stage we define expressions to be agent *names*, messages sends, or *paths* that define composite expressions. Message sends are denoted by $a \texttt{ send } m(\vec{A}_n)$, where a is the name of an agent, and $m(\vec{A}_n)$ invokes the m method found in a with arguments $\vec{A}_n$. 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 2.3.

2.2 Structural Congruence

Structural congruence for agents is defined in terms of a relation of equivalence over method sequences, given in Figure 2. The intention is to allow method suites to be reordered without affecting the behavior of the enclosing agent. Definitions for methods

$(L, M), N \leftrightharpoons L, (M, N)$	(Eq Meth Assoc)
$n \neq m \Rightarrow M, m(x) \triangleright P; n(y) \triangleright Q \leftrightharpoons M, n(y) \triangleright Q, m(x) \triangleright P$	(Eq Meth Comm)
$M, m(x) \triangleright P, m(x) \triangleright Q, M \leftrightharpoons M, m(x) \triangleright Q$	(Eq Meth Over)
$M \leftrightharpoons M$	(Eq Meth Refl)
$M \leftrightharpoons N \Rightarrow N \leftrightharpoons M$	(Eq Meth Symm)
$L \leftrightharpoons M, M \leftrightharpoons N \Rightarrow L \leftrightharpoons N$	(Eq Meth Trans)

Fig. 2. Equivalence for Methods

with different name and/or arity maybe freely permuted; instead, if the same method has multiple definitions, then the rightmost definition overrides the remaining ones. Similar notions of equivalence between method suites can be found in the literature on objects: in fact, our definition is directly inspired by the bookkeeping relation introduced in [8].
The relation structural congruence of processes is defined as the smallest congruence

$(\nu x)\mathbf{0} \equiv \mathbf{0}$	(Struct Res Dead)
$x \neq y \Rightarrow (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$	(Struct Res Res)
$x \notin \mathit{fn}(P) \Rightarrow (\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$	(Struct Res Par)
$(A.A').P \equiv A.A'.P$	(Struct Path Assoc)
$x \notin \mathit{fn}(M) \cup \{a\} \Rightarrow (\nu x)a[M\,;P] \equiv a[M\,;(\nu x)P]$	(Struct Res Agent)
$\varepsilon.P \equiv P$	(Struct Empty Path)
$M \leftrightharpoons N \Rightarrow a[M\,;P] \equiv a[N\,;P]$	(Struct Cong Agent Meth)

Fig. 3. Structural Congruence for Agents

on processes that forms a commutative monoid with product $\mid$ and unit **0**, and is closed under the rules in Figure 3. These rules are parametric in the definition of expressions and *free names*, which vary depending on the specific calculus at issue (for MA^{++}, expressions are given in Section 3, while free names are defined by a standard extension of the definition in [3]).

The first block of clauses are standard (they are the rules of the π-calculus). The rule (Struct Path Assoc) is a structural equivalence rule for the Ambient Calculus, while the rule (Struct Res Agent) modifies the rule for agents in the Ambient and Seal calculi to account for the presence of methods. The last rule establishes agent equivalence up to reordering of method suites. In addition, we identify processes up to renaming of bound variables: $(\nu p)P = (\nu q)P\{p := q\}$ if $q \notin \mathit{fn}(P)$. The behavior of agents is now defined in terms of a reduction relation which obeys the structural rules in Figure 4, plus specific rules for each of the construct in the calculus.

(**Red Struct**)	$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \quad \Rightarrow \quad P' \rightarrow Q'$
(**Red Amb**)	$P \rightarrow Q \quad \Rightarrow \quad a[M\,;P] \rightarrow a[M\,;Q]$
(**Red Res**)	$P \rightarrow Q \quad \Rightarrow \quad (\nu x)P \rightarrow (\nu x)Q$
(**Red Par**)	$P \rightarrow Q \quad \Rightarrow \quad P \mid R \rightarrow Q \mid R$

Fig. 4. Structural Rules for Reduction

2.3 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 (i) where the method invocation should occur, and (ii) as to where the body of the invoked method should be executed. Below, we discuss two possible modes for invocation and, within each mode we illustrate two possible locations where the invoked process can be activated.

Remote Invocation. This mode arises in a situation where the sender and the receiver of the message are siblings and the message is sent by the process enclosed within the sender. Remote invocation is consistent with the "subjective" model of mobility, in which the moves of an object are regulated from within the object, by its controlling process: similarly, the object delegates to its controlling process the ability to communicate with its siblings.

Once the message is sent, the selected method can be activate either on the sender or on the receiver. There is a close analogy between these two possibilities, and two common protocols in distributed systems. Activating the method in the sender corresponds to the protocol known as *code on demand*.

(Code on Demand)

$$a[M\,; b\ \texttt{send}\ m(\vec{A}).P] \mid b[N, m(\vec{x}) \triangleright \varsigma(z)Q\,; R]$$
$$\rightarrow\ a[M\,; Q\{z := b, \vec{x} := \vec{A}\} \mid P] \mid b[N, m(\vec{x}) \triangleright \varsigma(z)Q\,; R]$$

Having requested a service, the sender takes the load of executing the corresponding process: this is a common practice for modern distributed systems, and specifically, for the Web. Upon receiving a request, a server spawns a new process (e.g., a Java *applet*) authorizing the client to execute it: the process is activated on the client side to not overburden the server with computations loads pertaining to its clients.

The alternative is to activate the method body on the receiver side, thus mimicking a *remote procedure call* (or more precisely a *remote method invocation*).

(Remote Procedure Call)

$$a[M\,; P \mid b\ \texttt{send}\ m(\vec{A}).P'] \mid b[M', m(\vec{x}) \triangleright \varsigma(z)Q\,; R]$$
$$\rightarrow\ a[M\,; P \mid P'] \mid b[M', m(\vec{x}) \triangleright \varsigma(z)Q\,; Q\{z := b, \vec{x} := \vec{A}\} \mid R]$$

This alternative is just as reasonable. There is, however, a technical argument in favor of our first solution: activating the method body within the receiver makes it difficult to give a uniform reduction for messages to siblings and messages to *self*. Consider the following example:

$$a[M\,; b\ \texttt{send}\ m_1.P] \mid b[m_1 \triangleright \varsigma(z)z\ \texttt{send}\ m_2, m_2 \triangleright Q\,;]$$

If the method body is activated on the sender, the configuration reduces to $a[M\,; P \mid Q]$, as expected, in two steps. If instead, the body of m_1 is activated on the receiver, then one reduction leads to the configuration

$$a[M\ ;\ P] \mid b[m_1 \triangleright \varsigma(z) z \ \texttt{send}\ m_2, m_2 \triangleright Q\ ;\ b\ \texttt{send}\ m_2].$$

At this stage is no sibling object b to which the message m_2 may be sent. Although the problem may easily be remedied using special syntax (and reduction) for local method calls, our first solution is formally simpler and more elegant.

Local Invocation In this mode, the message is sent from an agent to one of its children: the invocation is *local* in that it does not extrude the scope of the sender. This mode is consistent with the "objective" model of mobility, in which the moves of an object are regulated by the enclosing environment: similarly, the environment requests the services provided by its enclosed objects.

The activation of the selected method may take place either at the same level as the invocation, or in the receiver. In the first case, the reduction is

$$\begin{aligned} & b\ \texttt{send}\ m(\vec{A}).P \mid b[M; m(\vec{x}) \triangleright \varsigma(z)Q\ ;\ R] \\ & \qquad\qquad \rightarrow\ P \mid Q\{z := b, \vec{x} := \vec{A}\} \mid b[M; m(\vec{x}) \triangleright \varsigma(z)Q\ ;\ R] \end{aligned}$$

Having requested a service to a child, the environment takes the load of executing the corresponding process. As for the case of remote invocation, the method body could be activated within the receiver, but again the definition of reduction would not be uniform for messages to children and messages to *self*.

This concludes the discussion on the general model. In the next section we illustrate an instance of the model, based on Mobile Ambients [3, 5].

3 MA^{++}

As we anticipated, our mobile objects combine the functionalities of Mobile Ambients and objects. We will use the terms object and ambient interchangeably to refer to agents in MA^{++}: the context will prevent any source of confusion between our agents and agents from MA.

Processes and methods are defined as in Figure 1, while the (now complete) syntax of expressions is given by the productions below:

$$\begin{aligned} A\ ::=\ & x \mid a\ \texttt{send}\ m(\vec{A}) \mid \varepsilon \mid A.A \\ & \mid \texttt{in}\ a \mid \texttt{out}\ a \mid \texttt{open}\ a \end{aligned}$$

The `in` and `out` expressions provide objects with the same mobility capabilities as ambients, while `open` allows an object to break through the boundary of its enclosed objects and incorporate their body. As anticipated, the calculus does not include any construct or resource for synchronization: objects only communicate via messages, and message sends are synchronous. As we shall illustrate, different forms of process communication can be encoded in terms of the existing constructs.

3.1 Reduction Semantics

The definition of structural congruence and structural reduction are inherited directly from the general model. In Figure 5, we give the reduction rules distinctive of ambients in MA^{++}.

(in)	$a[M\,;\,\mathtt{in}\ b.P \mid Q] \mid b[N\,;\,R] \rightarrow b[N\,;\,R \mid a[M\,;\,P \mid Q]]$	
(out)	$a[M\,;\,b[N\,;\,\mathtt{out}\ \ a.P \mid Q] \mid R] \rightarrow b[N\,;\,P \mid Q] \mid a[M\,;\,R]$	
(open$_1$)	$\mathtt{open}\ \ a.P \mid a[Q] \rightarrow P \mid Q$	
(open$_2$)	$b[M\,;\,\mathtt{open}\ \ a.P \mid a[N\,;\,Q] \mid R] \rightarrow b[M, N\,;\,P \mid Q \mid R]$	for $N \neq \varepsilon$
(send)	$a[M\,;\,P \mid b\ \mathtt{send}\ m(\vec{A}).R] \mid b[N, m(\vec{x}) \triangleright \varsigma(z)Q\,;\,S]$ $\rightarrow a[M\,;\,P \mid Q\{z, \vec{x} := b, \vec{A}\} \mid R] \mid b[N, m(\vec{x}) \triangleright \varsigma(z)Q\,;\,S]$	

Fig. 5. Reduction Rules for Ambients

The reduction rule **(send)** implements the remote mode for code-on-demand model we discussed in the previous section. The reduction rules for the `in` and `out` actions are defined exactly as in the Ambient Calculus. The action `in` $a.P$ instructs the ambient surrounding `in` $a.P$ to enter a sibling ambient named a. If no sibling named a exists, the operation blocks. The action `out` $a.P$ instructs the ambient surrounding `out` $a.P$ to exit its parent ambient named a: if the parent is not named a, the operation blocks until a time when such a parent exists.

The reduction for `open` depends on whether the method suite of the opened ambient is empty or not. If it is empty, the reduction is exactly the same as in the Ambient Calculus: `open` a dissolves the boundary of an ambient named a located at the same level as `open`, unleashing the process enclosed in a. If instead a contains a nonempty method suite, `open` a may only be reduced within an enclosing ambient and its effect is twofold: besides unleashing the process contained in a, as in the previous case, it also merges the method suites of the opening and opened ambients. In both cases, if no ambient named a exists, `open` a blocks. As we show below, this behavior of open allows an elegant encoding of the operations of method update available in object calculi.

4 Examples

Parent-Child Messages. Having chosen the remote mode for method invocation, it is often useful for an ambient to be able to send messages to its parent or its children. We denote the two forms of communication as follows:

$a^{\downarrow}\ \mathtt{send}\ m(\vec{A}).P$	send $m(\vec{A})$ to child a
$a^{\uparrow}\ \mathtt{send}\ m(\vec{A}).P$	send $m(\vec{A})$ to parent a

Both these invocation modes can be derived with the existing constructs. Parent-to-child invocation can be defined as follows:

$$a^{\downarrow}\ \mathtt{send}\ m(\vec{A}).P \quad \triangleq \quad (\nu p, q)(p[a\ \mathtt{send}\ m(\vec{A}).q[\mathtt{out}\ p]] \mid \mathtt{open}\ q.\mathtt{open}\ p.P)$$

where $p, q \notin \mathit{fn}(\vec{A}) \cup \mathit{fn}(P)$. Then, it is easy to verify that:

$$a^{\downarrow}\,\mathtt{send}\, m(\vec{A}).P \mid a[m(\vec{x}) \triangleright \varsigma(z)Q; R] \;\; \longrightarrow^{*} \;\; P \mid Q\{z,\vec{x} := a,\vec{A}\} \mid a[m(\vec{x}) \triangleright \varsigma(z)Q; R]$$

Child-to-parent invocation can be defined similarly, as follows:

$$a^{\uparrow}\,\mathtt{send}\; m(\vec{A}).P \triangleq (\nu p, q)(\mathtt{open}\; q.\mathtt{open}\; p \mid p[\mathtt{out}\; a.a\;\mathtt{send}\; m(\vec{A}).\mathtt{in}\; a.q[\mathtt{out}\; p.P]])$$

where $p, q \notin \mathit{fn}(\vec{A}) \cup \mathit{fn}(P)$. Then $a[M, m(\vec{x}) \triangleright \varsigma(z)Q \,;\, a^{\uparrow}\;\mathtt{send}\; m(\vec{A}).P \mid R]$ reduces after some steps to $(\nu p, q)(a[M, m(\vec{x}) \triangleright \varsigma(z)Q \,;\, Q\{z, \vec{x} := a, \vec{A}\} \mid P \mid R])$ as expected.

Replication. The behavior of replication in concurrent calculi is typically defined by a structural equivalence rule establishing that $!P \equiv !P \mid P$. With ambients, we can encode a similar construct relying upon the implicit form of recursion inherent in the reduction of method invocation. The coding is as follows:

$$!P \;\triangleq\; (\nu p)(p[\mathit{bang} \triangleright \varsigma(z)z^{\downarrow}\;\mathtt{send}\;\mathit{bang} \mid P \,;\,] \mid p^{\downarrow}\;\mathtt{send}\;\mathit{bang})$$

where $p \notin \mathit{fn}(P)$. Using the derived reduction rule for downward method invocation, for every P, $!P$ reduces in one (encoded) step to $P \mid !P$ as desired. Similarly we can encode guarded replication $!A.P$ —where replication is performed only after the consumption of A— as follows:

$$(\nu p)(A.p^{\downarrow}\;\mathtt{send}\;\mathit{bang} \mid p[\mathit{bang} \triangleright \varsigma(z)A.z^{\downarrow}\;\mathtt{send}\;\mathit{bang} \mid P \,;\,])$$

Method update. Following the standard definition of method override [1, 8] method updates for ambients can be formulated, informally, as follows: given the ambient $a[M; m(\vec{x}) \triangleright P \,;\, Q]$ we wish to replace the current definition P of $m(\vec{x})$ by the new definition P' to form the ambient $a[M; m(\vec{x}) \triangleright P' \,;\, Q]$.

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

Updates as processes: Update processes are denoted by $x \cdot m(\vec{y}) \triangleright \varsigma(s)P$, read "the m method at x gets definition P ". We define their behavior as follows: let first

$$a \cdot m(\vec{x}) \triangleright \varsigma(s)P \;\triangleq\; \mathrm{UPD}[m(\vec{x}) \triangleright \varsigma(s)P \,;\, \mathtt{in}\;\; a].$$

Then define an updatable ambient as follows

$$a^{\star}[M \,;\, P] \;\triangleq\; a[M \,;\, !(\mathtt{open}\;\;\mathrm{UPD}) \mid P].$$

Now, if we form the composition $a \cdot m(\vec{x}) \triangleright \varsigma(z)P' \mid a^{\star}[M \,;\, m(\vec{x}) \triangleright P \,;\, Q]$, the reduction for $\mathtt{open}$ enforces the expected behavior:

$$a \cdot m(\vec{x}) \triangleright P' \mid a^{\star}[M \,;\, m(\vec{x}) \triangleright P \,;\, Q] \quad \longrightarrow^{*} \quad a^{\star}[M \,;\, m(\vec{x}) \triangleright P' \,;\, Q]$$

Multiple updates for the same method may occur in parallel, in which case their relative order is established nondeterministically. The coding works well also with "self inflicted" updates: for example, the configuration

$$a^{\downarrow}\;\mathtt{send}\; m.P \mid a^{\star}[m \triangleright \varsigma(z)z \cdot n(\vec{x}) \triangleright Q', n(\vec{x}) \triangleright Q \,;\, R]$$

reduces to

$$P \mid a^{\star}[m \triangleright \varsigma(z) z \cdot n(\vec{x}) \triangleright Q', n(\vec{x}) \triangleright Q' \,;\, R]$$

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

Updates as expressions: Sequential updates are defined similarly to update processes. In this case, $(a \cdot m(\vec{x}) \triangleright \varsigma(s)P).Q$ first updates m at a 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:

$$(a \cdot m(\vec{y}) \triangleright \varsigma(z)P).Q \ \stackrel{\triangle}{=}\ (\nu p)(\text{UPD}[m(\vec{y}) \triangleright \varsigma(z)P \,;\, \texttt{in}\ a.p[\,;\, \texttt{out}\ a.Q]] \mid \texttt{open}\ p)$$

where $p \notin fn(P \mid Q)$. Then we have:

$$(a \cdot m(\vec{x}) \triangleright P').Q \mid a^{\star}[M, m(\vec{x}) \triangleright P \,;\, R] \quad \to^{*} \quad Q \mid a^{\star}[M, m(\vec{x}) \triangleright P' \,;\, R]$$

Process communication The next example shows that synchronous and asynchronous communication primitives between processes can be encoded. We first give an encoding of synchronous communication. A similar model of (asynchronous) channel-based communication is presented in [5] and it is based on the more primitive form of local and *anonymous* communication defined for the Ambient Calculus: here, instead, we rely on the ability, distinctive of our ambients, to exchange values between methods.

A channel n is modeled by a (parallel composition of) an updatable ambient n, and two locks n^i, and n^o. The ambient n contains a method *msg*: a process willing to read from n installs itself as the body of this method, whereas a process willing to write on n invokes *msg* passing along the argument of the communication.

$$\begin{aligned}
(ch\ n) &\stackrel{\triangle}{=} n^{\star}[msg(x) \triangleright \mathbf{0}] \mid n^i[\,] \\
n\langle A\rangle.Q &\stackrel{\triangle}{=} \texttt{open}\ n^o.n^{\downarrow}\ \texttt{send}\ msg(A).(n^i[\,] \mid Q) \\
n(x).P &\stackrel{\triangle}{=} \texttt{open}\ n^i.n \cdot msg(x) \triangleright \varsigma(z)P.n^o[\,] \qquad (z \notin fn(P))
\end{aligned}$$

The communication protocol is as follows: A process $n\langle A\rangle.Q$ writing A on n first attempts to grab the output lock n^o, then sends the message $msg(A)$ to n, and finally continues as Q releasing the input lock n^i. At the start of the protocol there are no output locks: hence the process writing on n blocks. A process $n(x).P$ reading from n first grabs the input lock n^i provided by the channel, then installs itself as the body of the *msg* method in n, and finally releases the output lock. Now the writing process resumes its computation: it sends the message thus unleashing P, and then releases the input lock and continues as Q.

Asynchronous communications are obtained directly from the coding above, by a slight variation of the definition of $n\langle A\rangle.Q$. We simply need a different parenthesizing:

$$n\langle A\rangle.Q \ \stackrel{\triangle}{=}\ (\texttt{open}\ n^o.n^{\downarrow}\ \texttt{send}\ msg(A).(n^i[])) \mid Q$$

Based on the this technique, we can encode the synchronous (and similarly, the asynchronous) π-calculus in ways similar to what is done in [6]. Each name n in the π-calculus becomes a triple of names in our calculus: the name n of the ambient dedicated to the communication, and the names n^i and n^o of the two locks. Therefore, communication of a π-calculus name becomes the communication of a triple of ambient names.

$$\begin{array}{rcll}
\llbracket\,(\nu n)P\,\rrbracket & \triangleq & (\nu n, n^i, n^o)(n^i[\,] \mid n^\star[\mathit{msg}(x, x^i, x^o) \triangleright \mathbf{0}] \mid \llbracket\,P\,\rrbracket\,) & n^i, n^o \notin \mathit{fn}(\llbracket\,P\,\rrbracket\,) \\
\llbracket\,n\langle y\rangle.Q\,\rrbracket & \triangleq & \texttt{open}\ n^o.n^\downarrow\ \texttt{send}\ \mathit{msg}(y, y^i, y^o).(n^i[\,] \mid Q) & \\
\llbracket\,n(x).P\,\rrbracket & \triangleq & \texttt{open}\ n^i.n \cdot \mathit{msg}(x, x^i, x^o) \triangleright \varsigma(z)P.n^o[\,] & \\
\llbracket\,P \mid Q\,\rrbracket & \triangleq & \llbracket\,P\,\rrbracket \mid \llbracket\,Q\,\rrbracket & \\
\llbracket\,!P\,\rrbracket & \triangleq & (\nu n)(n[\mathit{bang} \triangleright \varsigma(z)z^\downarrow\ \texttt{send}\ \mathit{bang} \mid \llbracket\,P\,\rrbracket\ ;\,] \mid n^\downarrow\ \texttt{send}\ \mathit{bang}) & n \notin \mathit{fn}(\llbracket\,P\,\rrbracket\,)
\end{array}$$

Fig. 6. Encoding of the synchronous π-calculus

The initialization of the *msg* method in the ambient that encodes the channel n could be safely omitted, without affecting the operational properties of encoding. However, as given, the definition scales smoothly to the case of a typed encoding, preserving well-typing.

5 Types and Type Systems

The typing of ambients inherits ideas from existing type systems for Mobile Ambients: however, as we anticipated, the presence of methods enables a more structured (and informative) characterization of their enclosing ambient's interfaces. The productions defining the set of types are given below:

Signatures	Σ	$::=$	$m(\mathscr{W}) \triangleright \mathscr{P}, \Sigma \quad \mid \quad \varepsilon$
Ambients	$\mathscr{A}$	$::=$	$\texttt{Amb}[\Sigma]$
Capabilities	$\mathscr{C}$	$::=$	$\texttt{Cap}[\Sigma]$
Processes	$\mathscr{P}$	$::=$	$\texttt{Proc}[\Sigma]$
Values	$\mathscr{W}$	$::=$	$\mathscr{A} \mid \mathscr{C}$

Signatures convey information about the interface of an ambient, by listing the ambient's method names, input type as well as the type of the method bodies. The intuitive reading of ambient, capability and process types is as follows: the type $\texttt{Amb}[\Sigma]$ is the type of ambients with methods declared in Σ; the type $\texttt{Cap}[\Sigma]$ is the type of capabilities[1] whose enclosing ambient (if any) has a signature which contains at least the methods included in Σ; the type $\texttt{Proc}[\Sigma]$ is the type of processes whose enclosing ambient (if there is any) contains at least all the methods declared in Σ.

The essential novelty over previous type systems for Mobile Ambients [6, 4, 13] is that we use method signatures as tags for ambient and capability types: in [6], instead, ambient (and capability) types expose the type of values that can be exchanged as a result of local process communication. This difference reflects the different communication primitives in the two calculi: specifically, communication is accomplished via message sends in our calculus, whereas it relies on explicit input/output primitives in MA.

[1] *Capability* is the term used in [5] to refer to "actions": capabilities can be transmitted over channels, and transmitting a capability corresponds to trasmit the *capability* of performing the corresponding action. The same intuition justifies the use of the term in MA^{++}.

5.1 Type System

The typed syntax of the calculus is defined by the following productions[2] :

Methods	$M ::= m(x{:}\mathscr{W}) \triangleright \varsigma(z{:}\mathscr{A})P \mid M, M \mid \varepsilon$
Processes	$P ::= \mathbf{0} \mid P \vert P \mid a[M\,;P] \mid (\nu x{:}\mathscr{A})P \mid A.P$
Expressions	$A ::= x \mid a\ \mathtt{send}\ m(\vec{A}) \mid \mathtt{in}\ a \mid \mathtt{out}\ a \mid \mathtt{open}\ a \mid A.A \mid \varepsilon$

The type system derives three kinds of judgments (where E denotes generic expressions and processes): $\Gamma \vdash \diamond$ (well-formed type environment), $\Gamma \vdash E : T$ (typing), and $\Gamma \vdash T_1 \le T_2$ (subtyping). The typing and subtyping rules presented in Figure 7 are discussed below.

Method signatures, associated with ambient types, are traced by the types Cap, of capabilities, to allow an adequate typing of messages and mobility: specifically, the rule (OPEN) establishes that opening an ambient $a : \mathtt{Amb}[\Sigma]$ is legal under the condition that the signature of the opening ambient is equal to (in fact, contains, given the presence of subtyping) the signature of the ambient being opened. This condition is necessary, as subject reduction would otherwise fail: as a consequence, opening an ambient may only update existing methods of the opening ambient, and the update must preserve the types of the original methods.

Signatures are not traced when typing expressions involving moves or messages: for the latter, see rule (MESSAGE), the capability type has the same signature as the process type of the body of the invoked method. Of course, in order for the expression to type check the message argument and the method parameters must have the same type[3].

The typing of processes is standard (cf. [6, 13]), with the only exception of the rule (AMB) which defines the types of ambients. Ambients are typed similarly to objects in the object calculi of [1]: each method is typed under the assumptions that (i) the self parameter has the same type of the enclosing ambient, and (ii) that method parameters have the declared type. The condition $i \in \textsc{last}(I)$ (where $\textsc{last}(I)$ denotes the set $\{i \in I \mid \forall j > i, m_j \neq m_i\}$) ensures that only the rightmost definition of a method is considered when typing an ambient[4]. Finally, no constraint is imposed on the signature Σ', associated with the process type in the conclusion of the rule, as that signature is (a subset of) the signature of the ambient enclosing a (if any). As for the subtyping relation, non-trivial subtyping is defined for capability and process types: specifically, a capability (resp. process) type $\mathtt{Cap}[\Sigma]$ (resp. $\mathtt{Proc}[\Sigma]$) is a subtype of any capability (resp. process) type whose associated signature (set theoretically) contains Σ. The resulting notion of subtyping corresponds to the contravariant subtyping in *width* distinctive of variant types. The covariant width subtyping typical of object and record

[2] The other typed versions Ambients [6, 4, 13] allow restrictions on varibles of type $\mathscr{W}$ (rather than just $\mathscr{A}$), but we do not see the purpose of such a generalization.

[3] In fact, since capability and ambient types can be subtyped, the type of the arguments can be subtypes of the type of the formal parameters.

[4] Technically speaking, we need this restriction to ensure the subject reduction property since without it a well-typed term could be structurally equivalent (and, therefore, reduction equivalent) to an ill-typed one.

Type environments

(ENV-EMPTY)
$$\frac{}{\varnothing \vdash \diamond}$$

(ENV-NAME)
$$\frac{\Gamma \vdash \diamond \quad x \notin Dom(\Gamma)}{\Gamma, x : \mathscr{W} \vdash \diamond}$$

Expressions

(NAME/VAR)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)}$$

(PATH)
$$\frac{\Gamma \vdash A_1 : \mathtt{Cap}[\Sigma] \quad \Gamma \vdash A_2 : \mathtt{Cap}[\Sigma]}{\Gamma \vdash A_1.A_2 : \mathtt{Cap}[\Sigma]}$$

(OPEN)
$$\frac{\Gamma \vdash a : \mathtt{Amb}[\Sigma]}{\Gamma \vdash \mathtt{open}\ a : \mathtt{Cap}[\Sigma]}$$

(INOUT)
$$\frac{\Gamma \vdash a : \mathtt{Amb}[\Sigma] \quad (A' \in \{\mathtt{in}\ a, \mathtt{out}\ a\})}{\Gamma \vdash A' : \mathtt{Cap}[\Sigma']}$$

(MESSAGE)
$$\frac{\Gamma \vdash a : \mathtt{Amb}[\Sigma] \quad \Gamma \vdash A' : \mathscr{W} \quad (m(\mathscr{W}) \triangleright \mathtt{Proc}[\Sigma'] \in \Sigma)}{\Gamma \vdash a\ \mathtt{send}\ m(A') : \mathtt{Cap}[\Sigma']}$$

Processes

(PREF)
$$\frac{\Gamma \vdash A : \mathtt{Cap}[\Sigma] \quad \Gamma \vdash P : \mathtt{Proc}[\Sigma]}{\Gamma \vdash A.P : \mathtt{Proc}[\Sigma]}$$

(PAR)
$$\frac{\Gamma \vdash P : \mathtt{Proc}[\Sigma] \quad \Gamma \vdash Q : \mathtt{Proc}[\Sigma]}{\Gamma \vdash P \mid Q : \mathtt{Proc}[\Sigma]}$$

(RESTR)
$$\frac{\Gamma, x : \mathscr{A} \vdash P : \mathtt{Proc}[\Sigma]}{\Gamma \vdash (\nu x : \mathscr{A})P : \mathtt{Proc}[\Sigma]}$$

(DEAD)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \mathtt{Proc}[\Sigma]}$$

(AMB) $\quad (\Sigma = (m_i(\mathscr{W}_i) \triangleright \mathtt{Proc}[\Sigma_i])_{i \in I} \quad \mathscr{A} = \mathtt{Amb}[\Sigma] \quad i \in \textsc{last}(I))$
$$\frac{\Gamma \vdash A : \mathscr{A} \quad \Gamma, z : \mathscr{A}, x_i : \mathscr{W}_i \vdash P_i : \mathtt{Proc}[\Sigma_i] \quad \Gamma \vdash P : \mathtt{Proc}[\Sigma]}{\Gamma \vdash A[(m_i(x_i : \mathscr{W}_i) \triangleright \varsigma(z : \mathscr{A})P_i)_{i \in I}\ ;\ P] : \mathtt{Proc}[\Sigma']}$$

Subsumption

(SUBS)
$$\frac{\Gamma \vdash E : \mathscr{W} \qquad \mathscr{W} \leq \mathscr{W}'}{\Gamma \vdash E : \mathscr{W}'}$$

Subtyping

(SUBCAP)
$$\frac{\Sigma \subseteq \Sigma'}{\mathtt{Cap}[\Sigma] \leq \mathtt{Cap}[\Sigma']}$$

(SUBPROC)
$$\frac{\Sigma \subseteq \Sigma'}{\mathtt{Proc}[\Sigma] \leq \mathtt{Proc}[\Sigma']}$$

Fig. 7. Typing and Subtyping Rules

types must be disallowed over ambient types to ensure sound uses of the `open` capability: intuitively, when opening an enclosed ambient, we need *exact* knowledge of the contents of that ambient, (specifically, of its method suite) so as to ensure that all the overriding that takes place upon exercising the capability be traced in the types.

As customary, the subtyping relation is endowed in the type system via a subsumption rule.

5.2 Subject Reduction and Type Soundness

We conclude the description of the type system with a subject reduction theorem and a discussion on type soundness. The, rather standard, proof is only outlined here due to the lack of space.

Lemma 1 (Substitution). *If* $\Gamma, x{:}\mathscr{W} \vdash P : \mathtt{Proc}[\Sigma]$ *and* $\Gamma \vdash A{:}\mathscr{W}$*, then* $\Gamma \vdash P\{x := A\} : \mathtt{Proc}[\Sigma']$ *with* $\Sigma' \subseteq \Sigma$.

Proposition 1 (Subject Congruence).
1. If $\Gamma \vdash P : \mathtt{Proc}[\Sigma]$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q : \mathtt{Proc}[\Sigma]$.
2. If $\Gamma \vdash P : \mathtt{Proc}[\Sigma]$ *and* $Q \equiv P$ *then* $\Gamma \vdash Q : \mathtt{Proc}[\Sigma]$.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash P : \mathtt{Proc}[\Sigma]$ *and* $P \rightarrow Q$ *then* $\Gamma \vdash Q : \mathtt{Proc}[\Sigma']$ *with* $\Sigma' \subseteq \Sigma$.

Besides being interesting as a meta-theoretical property of the type system, subject reduction may be used to derive a soundness theorem ensuring the absence of run-time (type) errors for well-typed programs. As we anticipated, the errors we wish to detect are those of the kind "message not understood" distinctive of object calculi. With the current definition of the reduction relation such errors do not arise, as not-understood messages simply block: this is somewhat unrealistic, however, as the result of sending a message to an object (a server) which does not contain a corresponding method should be (and indeed is, in real systems) reported as an error. We thus introduce a new reduction to account for these situations:

$$a[M\,;\,P \mid b\ \mathtt{send}\ m(\vec{A}).Q] \mid b[N\,;\,R] \rightarrow a[M\,;\,P \mid \mathtt{ERR}] \mid b[N\,;\,R] \qquad (m \notin N)$$

The intuitive reading of the reduction is that a not-understood message causes a local error —for the sender of that message— rather than a global error for the entire system. The reduction is meaningful also in the presence of multiple ambients with equal name, as our type system (like those of [6, 4, 13]) ensures that ambients with the same name have also the same type. Hence, if a method m is absent from a given ambient b, it will also be absent from all ambients named b. If we take `ERR` to be a distinguished process, with no type, it is easy to verify that no system containing an occurrence of `ERR` can be typed in our type system. Absence of run-time errors may now be stated follows:

Theorem 2. *Let* P *be a well-typed MA*$^{++}$ *process. Then, there exist no context* $\mathbf{C}[-]$ *such that* $P \rightarrow^{*} \mathbf{C}[\mathtt{ERR}]$.

6 Extensions

There are several desirable extensions to MA^{++} and its type system. The most natural is the ability to treat method names as ordinary names. This would allow one to define private methods, and to give a formal account of dynamic messages. Both the extensions can be accommodated for free in the untyped calculus. For the typed version, instead, things are more complex. It is possible (and relatively easy) to extend the syntax and allow method names to be restricted. Instead, disallowing method names as values is more critical. The reason is that method names occur in the signatures of ambient (capability and process) types: consequently, allowing methods to be passed would be possible but it would make our types (first-order) dependent types (see [10] for similar restrictions).

A further extension has to do with Safe Ambients. In [13] the authors describe an extension of the calculus of Mobile Ambients, called Safe Ambients, where entering, exiting and opening an ambient requires a corresponding co-action by the ambient that undergoes the action. The use of co-actions allows (i) a more fine-grained control on when actions take place, and (ii) the definition of a refined type system where types can be used to essentially "serialize" the activities of the parallel processes controlling the moves of an ambient. As shown in [13], the combination of these features makes it possible to define a rich algebraic theory for the resulting calculus. The idea of co-actions and of single-threaded types can be incorporated in the type system we have described in the previous sections rather smoothly: besides the co-actions related to mobility, we simply need a co-action for messages, and a modified reduction rule for message sends that requires the receiver to be *listening* (i.e. to exercise the co-action corresponding to `send`) in order to reduce the message. We leave this as subject of future work.

Finally, it would be interesting to include linear types to ensure (local) absence of ambients with the same name: in fact, while the possibility of there being more than one ambient that is willing to receive a given message provides useful forms of nondeterminism, ensuring linearity of ambient names could be useful to prevent what [13] defines "grave interferences" and thus to prove interesting behavioral properties of method invocation.

7 Conclusions

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 MA^{++} we have described should be understood as a first step 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" [11].

More work is clearly needed to evaluate the adequacy of the calculus as a formal tool for modeling realistic applications, to develop a reasonable algebraic theory for the calculus, and to study techniques of program static analysis other than typing.

All these aspects are current topic of research for the two calculi —Ambients and Seals— we had in mind when developing the general model. The recent papers on typed formulations of Ambients [6, 4, 13] 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 G. Castagna. Mobile objects. In *FOOL'7 Proc. of the 7th Int. Workshop on Foundations of Object Oriented Languages*. 2000. Electronic Proceedings.
3. L. Cardelli. Abstractions for mobile computations. In *Secure Internet Programming*, number 1603 in LNCS, 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 LNCS, pages 230–239. Springer, 1999.
5. L. Cardelli and A. Gordon. Mobile ambients. In *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. P Di Blasio and K. Fisher. A calculus for concurrent objects. In *CONCUR'96*, number 1119 in LNCS, pages 655–670. Springer, 1996.
8. 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.
9. 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.
10. Ms Hennessy and J. Riely. Resource access control in systems of mobile agents (extended abstract). In *Proc. of 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*. 1998.
11. D. Johansen. Trend wars. *IEEE Concurrency*, 7(3), Sept 1999.
12. J. Kleist and Sangiorgi D. Imperative objects and mobile processes. Unpublished manuscript.
13. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'2000*, pages 352–364. ACM Press, 2000.
14. U Nestmann, H. Huttel, J. Kleist, and M. Merro. Aliasing models for object migration. In *Proceedings of Euro-Par'99*, number 1685 in LNCS, pages 1353–1368. Springer, 1999.
15. 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 LNCS, pages 187–215. Springer-Verlag, April 1995.
16. V.T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *ECOOP '94*, number 821 in LNCS, pages 100–117. Springer, 1994.
17. J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, number 1686 in LNCS. Springer, 1999.
18. D.J Walker. Objects in the π calculus. *Information and Computation*, 116(2):253–271, 1995.