In ICTCS 2001, Lecture Notes in Computer Science, 2202:235–255, © Springer, 2001.

Subtyping and Matching for Mobile Objects ⋆

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

[1] Dipartimento di Informatica
Univ. "Ca' Foscari", Venezia, Italy

[2] Département d'Informatique
École Normale Supérieure, Paris, France

Abstract. In [BCC00], we presented a general framework for extending calculi of mobile agents with object-oriented features, and we studied a typed instance of that model based on Cardelli and Gordon's Mobile Ambients. Here, we refine our previous work and define a new calculus which is based on *Remote Procedure Call* as the underlying protocol for method invocation, and a different technique to type method bodies. The new type system is equipped with both a subtyping and a matching relation. The combination of matching and subtyping provides new insight into the relationship between ambient opening in the new calculus and method overriding in object-oriented calculi.

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. *Agents* are effective entities that perform computation and interact with other agents: the term "mobile" implies that agents are bound to *locations* and that this binding may vary over time; agent interaction, in turn, is achieved using resources such as communication channels

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 preliminary work on the subject [BC00,BCC00], in this paper we study a formal calculus that integrates object-oriented constructs into calculi of mobile agents. The resulting framework provides foundations for a computation model for distributed applications, where conventional client-server technology —based on remote exchange of messages between static sites— and mobile agents coexist in a uniform way.

The model results from extending the structure of *named* agents in the style of Mobile Ambients [CG98] with method definitions and primitives for dealing with message passing and self denotations. The extension has 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.

⋆ Work partially supported by MURST Project 9901403824_003, by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet", and by Galileo Action n. 02841UD

With respect to our previous work [BC00,BCC00] this paper brings two main contributions: the introduction of a Remote Procedure Call (RPC)[1] primitive for message passing and method invocation and, foremost, a non trivial blend of matching and subtyping relations. Method invocation based on RPC fits nicely the design of a typed distributed calculus as it allows method bodies to be type checked locally, in the object where they are defined, independently of the caller. As a consequence, the choice of RPC as the underlying semantics of method invocation yields a notion of interface type for our mobile objects that is substantially simpler and more tractable than the corresponding notion defined in [BC00,BCC00]. Matching is employed in the type system to ensure sound typing of ambient (or object) opening in the presence of methods residing within objects[2]. As it turns out, the combination of subtyping and matching conveys new insight into the relationship between method overriding in object-oriented calculi and the `open` capability in our mobile objects.

Plan of the paper In Section 2 we describe the calculus of mobile objects, named MA^{++} , based on the calculus of *Mobile Ambients* (henceforth MA) of [Car99,CG98]. Section 3 describes various examples of the expressive power of the calculus. Specifically, we show that it is possible to encode primitives like method overriding distinctive of object calculi, various forms of process communication, as well as different primitives of method execution[3]. In Section 4 we study the type theory of our calculus, and state relevant properties. Possible further extension are discussed in Section 5. The presentation of related work in Section 6, and final remarks in Section 7 conclude our presentation.

2 MA^{++}

2.1 Syntax

The syntax of MA^{++} is the same as that of mobile objects defined in [BC00,BCC00], and it results from generalizing the structure of ambients to include *interfaces*, as in $a[\![\, \mathrm{I}\,;\, P \,]\!]$, where P is a process and I is a list of method definitions, defined by the following productions:

$$
\begin{array}{llcll}
\text{Processes} & P & ::= & \mathbf{0} & \text{inactivity} \\
 & & \mid & P \mid P & \text{parallel composition} \\
 & & \mid & a[\![\, \mathrm{I}\,;\, P \,]\!] & \text{ambient} \\
 & & \mid & (\boldsymbol{\nu} x)P & \text{restriction} \\
 & & \mid & M.P & \text{action}
\end{array}
$$

[1] Remote Proceduce Call is often referred to as *Remote Method Invocation* (RMI) in this context.

[2] Note that under this aspect the type system in [BCC00] contained a flaw.

[3] Here, and throughout the paper, we use the terms “encode” and “encoding” in a 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.

$$
\begin{array}{llcll}
\text{Interfaces} & \mathrm{I} & ::= & \ell(\boldsymbol{x}) \triangleright \varsigma(z)P & \text{method} \\
 & & \mid & \mathrm{I} :: \mathrm{J} & \text{sequence} \\
 & & \mid & \varnothing & \text{empty interface} \\
 & & & & \\
\text{Patterns} & \boldsymbol{x} & ::= & x & \text{variable} \\
 & & \mid & (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) & \text{tuple } (n \geqslant 1)
\end{array}
$$

The syntax of processes is a generalization of the combinatorial kernel of the Ambient Calculus: $\mathbf{0}$ denotes the inactive process, $P \mid Q$ the parallel composition of two processes P and Q, $a[\![\mathrm{I}\,;P]\!]$ denotes the object named a with interface I and enclosed process P, $(\boldsymbol{\nu} x)P$ restricts the name x to P, and finally $M.P$ performs the action described by the term M and then continues as P.

Interfaces are lists of labels with associated processes: the syntactic form $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$ denotes a method labeled ℓ whose associated body is the process P where the ς-bounded variable z represents the *self* parameter distinctive of object calculi, i.e. the method's host object. Finally, the pattern $\boldsymbol{x}$ is the tuple of input parameters for P.

$$
\begin{array}{llcll}
\text{Terms} & M, N & ::= & a, b, \ldots, x, y \ldots & \text{name/variable} \\
 & & \mid & (M_1, \ldots, M_n) & \text{tuple } (n \geqslant 0) \\
 & & \mid & M.M & \text{path} \\
 & & \mid & \varepsilon & \text{empty path} \\
 & & \mid & \mathsf{in}\ a & \text{enter } a \\
 & & \mid & \mathsf{out}\ a & \text{exit } a \\
 & & \mid & \mathsf{open}\ a & \text{open } a \\
 & & \mid & a\ \mathsf{send}\ \ell\langle M \rangle & \text{remote invocation}
\end{array}
$$

Terms include the capabilities distinctive of Mobile Ambients. In addition, mobile objects are equipped with a capability for remote method invocation: $a\ \mathsf{send}\ \ell\langle M \rangle$ invokes the method labeled ℓ residing on the object denoted by a with arguments M.

In the following we use $P, Q, R, \ldots$ to range over processes, I, J to range over (possibly empty) interfaces, and lower case letters to range over generic names, preferring when possible $a, b, \ldots$ for agent names, and $x, y, \ldots$ for parameters. Method names, denoted ℓ range over a disjoint alphabet and have a different status: they are fixed labels that may not be restricted, abstracted upon, nor passed as values (they are similar to field labels in record-based calculi). We omit trailing or isolated $\mathbf{0}$ processes and empty interfaces, using M, $a[\![\mathrm{I}]\!]$, $a[\![P]\!]$, and $a[\![\ ]\!]$ as shorthands for, respectively, $M.\mathbf{0}$, $a[\![\mathrm{I}\,;\mathbf{0}]\!]$, $a[\![\varnothing\,;P]\!]$, and $a[\![\varnothing\,;\mathbf{0}]\!]$.

2.2 Operational Semantics

We define the operational semantics of the calculus by means of a structural congruence and a reduction relation. As usual, the former is used to rearrange a term in order to apply the latter.

Structural Congruence Structural congruence for agents is defined in terms of an equivalence relation $\equiv_{\mathrm{I}}$ over interfaces, given in Figure 1. This relation allows method suites to be reordered without affecting the behavior of the enclosing agent: reordering of methods, in turn, is used to define the reduction of method invocation.

(Eq Meth Assoc)	$(\mathrm{I} :: \mathrm{J}) :: \mathrm{L}$	$\equiv_{\mathrm{I}} \mathrm{I} :: (\mathrm{J} :: \mathrm{L})$	
(Eq Meth Comm)	$\mathrm{I} :: m(\boldsymbol{x}_m) \triangleright P; \ell(\boldsymbol{y}_\ell) \triangleright Q$	$\equiv_{\mathrm{I}} \mathrm{I} :: \ell(\boldsymbol{y}_\ell) \triangleright Q :: m(\boldsymbol{x}_m) \triangleright P$	$\ell \neq m$
(Eq Meth Over)	$\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright P :: \ell(\boldsymbol{x}) \triangleright Q :: \mathrm{I}$	$\equiv_{\mathrm{I}} \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright Q$	

Fig. 1. Equivalence for Methods

Definitions for methods with different name and/or arity maybe freely permuted (Eq Meth Comm); instead, if the same method has multiple definitions, then the right-most definition overrides the remaining ones (Eq Meth Over). 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 [FHM94].

Structural congruence of processes is defined as the smallest congruence on processes that forms a commutative monoid with product $\mid$ and unit $\mathbf{0}$, and is closed under the rules in Figure 2, where the set *fn* of *free names* is defined by a standard extension of the definition in [Car99].

(Struct Res Dead)	$(\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0}$	
(Struct Res Res)	$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$	$x \neq y$
(Struct Res Par)	$(\boldsymbol{\nu} x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu} x)Q$	$x \notin fn(P)$
(Struct Res Agent)	$(\boldsymbol{\nu} p)a[\![\mathrm{I}\,;\,P]\!] \equiv a[\![\mathrm{I}\,;\,(\boldsymbol{\nu} p)P]\!]$	$p \notin fn(\mathrm{I}) \cup \{a\}$
(Struct Path Assoc)	$(M.M').P \equiv M.M'.P$	
(Struct Empty Path)	$\varepsilon.P \equiv P$	
(Struct Cong Agent Meth)	$\mathrm{I} \equiv_{\mathrm{I}} \mathrm{J} \Rightarrow a[\![\mathrm{I}\,;\,P]\!] \equiv a[\![\mathrm{J}\,;\,P]\!]$	

Fig. 2. Structural Congruence for Agents

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 calculus to account for the presence of methods. Rule (Struct Cong Agent Meth) establishes agent equivalence up to reordering of method suites. In addition, we identify processes up to renaming of bound names: $(\boldsymbol{\nu} p)P = (\boldsymbol{\nu} q)P\{p := q\}$ if $q \notin fn(P)$.

Reduction Relation The reduction semantics of the calculus is given by the context rules in Figure 3, plus the notions of reduction collected in Figure 4, that we comment below

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

Fig. 3. Structural Rules for Reduction

(*in*)	$b[\mathrm{I}\,;\ \mathsf{in}\ \ a.P \mid Q] \mid a[\mathrm{J}\,;\,R] \ \rightarrow\ a[\mathrm{I}\,;\,R \mid b[\mathrm{J}\,;\,P \mid Q]]$
(*out*)	$a[\mathrm{I}\,;\,b[\mathrm{J}\,;\,\mathsf{out}\ \ a.P \mid Q] \mid R] \ \rightarrow\ b[\mathrm{J}\,;\,P \mid Q] \mid a[\mathrm{I}\,;\,R]$
(*open*)	$\mathsf{open}\ \ a.P \mid a[Q] \ \rightarrow\ P \mid Q$
(*update*)	$b[\mathrm{I}\,;\,\mathsf{open}\ \ a.P \mid a[\mathrm{J}\,;\,Q] \mid R] \ \rightarrow\ b[\mathrm{I} :: \mathrm{J}\,;\,P \mid Q \mid R]$ for $\mathrm{J} \neq \varepsilon$
(*send*)	$b[\mathrm{I}\,;\,a\ \mathsf{send}\ \ell\langle M\rangle.P \mid Q] \mid a[\mathrm{J} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,;\,S]$ $\rightarrow\ b[\mathrm{I}\,;\,P \mid Q] \mid a[\mathrm{J} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,;\,R\{z, \boldsymbol{x} := a, M\} \mid S]$

Fig. 4. MA^{++} reduction rules

The first three rules are exactly the same as the corresponding rules for the Mobile Ambients. Rule (*update*) is a direct generalization of the open rule to handle the case when the opened ambient contains a non-empty interface. In this case, $\mathsf{open}\ a$ may only be reduced within an enclosing ambient. After the opening, the local process of a is unleashed within b and the interfaces of the opening and the opened ambients are merged. The final (*send*) rule handles the new syntactic construct for method invocation, implementing the Remote Procedure Call (more precisely, Remote Method Invocation) model. The notation $R\{z, \boldsymbol{x} := a, M\}$ indicates simultaneous substitution in R of a for z and of M for $\boldsymbol{x}$. The result of the ambient b sending message ℓ to its sibling a, is thus the activation of the corresponding method body on the receiver side where actual parameters are substituted for formal ones and the *self* parameter is dynamically bound to the (name of the) receiver.

3 Expressive power

In this section, we discuss a number of protocols and constructs that can be expressed within MA^{++}. Some of these examples have been already presented in our previous work [BC00,BCC00] where, however, they were defined in terms of a different semantics for method invocation based on *Code On Demand*.

3.1 Parent-child and Local communications

Having chosen *Remote Procedure Call* as our primitive protocol, it is interesting to try to encode other alternatives, and see how an object could send messages to its parent or its children, as well as to invoke its own methods as shown in Figure 5. As it turns out, these alternative invocation modes can all be encoded.

$$
\begin{array}{ll}
(\textit{downsend}) & a\ \mathsf{downsend}\ \ell\langle M\rangle.P \mid a[\![\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R]\!] \\
& \quad \rightarrow\ P \mid a[\![\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R \mid Q\{z := a, \boldsymbol{x} := M\}]\!] \\
(\textit{upsend}) & a[\![\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R \mid b[\![\mathrm{J}\,;\, a\ \mathsf{upsend}\ \ell\langle M\rangle.P]\!]\,]\!] \\
& \quad \rightarrow\ a[\![\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R \mid Q\{z := a, \boldsymbol{x} := M\} \mid b[\![\mathrm{J}\,;\, P]\!]\,]\!] \\
(\textit{local}) & a[\![\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, a\ \mathsf{local}\ \ell\langle M\rangle.P_1 \mid P_2]\!] \\
& \quad \rightarrow\ a[\![\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, Q\{z, \boldsymbol{x} := M, a\} \mid P_1 \mid P_2]\!]
\end{array}
$$

Fig. 5. Other Constructs for Method Invocation

Parent-to-child invocation. This form of method invocation can be defined as follows:

$$a\ \mathsf{downsend}\ \ell\langle M\rangle.P \ \stackrel{\triangle}{=}\ (\boldsymbol{\nu} p, q)\,(p[\![\,a\ \mathsf{send}\ \ell\langle M\rangle.q[\![\mathsf{out}\ p]\!]\,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P)$$

where $(p, q \notin fn(M) \cup fn(P))$. In words, we temporarily create a new ambient p that becomes a sibling of the receiver in order to perform a RPC; we then use the ambient q as a "lock", to guarantee that the ambient p is destroyed only after the receiver has served the invocation. It is a routine check to verify that the desired effect of the invocation is achieved by a sequence of reduction steps. To ease the notation, we give the reduction steps in the simplified case of a method which does not have parameters and does not depend on *self* (the presence of parameters and the dependency on *self* does not interfere with the protocol).

$$
\begin{array}{l}
a\ \mathsf{downsend}\ \ell.P \mid a[\![\,\ell \triangleright Q\,;\, R]\!] \\
\quad \equiv (\boldsymbol{\nu} p, q)\left(p[\![\,\underline{a\ \mathsf{send}\ \ell\langle M\rangle}.q[\![\mathsf{out}\ p]\!]\,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P\right) \mid a[\![\,\ell \triangleright Q\,;\, R]\!] \\
\quad \rightarrow (\boldsymbol{\nu} p, q)\left(p[\![\,q[\![\underline{\mathsf{out}\ p}]\!]\,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P\right) \mid a[\![\,\ell \triangleright Q\,;\, R \mid Q]\!] \\
\quad \rightarrow (\boldsymbol{\nu} p, q)\left(p[\![\,]\!] \mid q[\![\,]\!] \mid \underline{\mathsf{open}\ q.\mathsf{open}\ p}.P\right) \mid a[\![\,\ell \triangleright Q\,;\, R \mid Q]\!] \\
\quad \rightarrow^{*} P \mid a[\![\,\ell \triangleright Q\,;\, R \mid Q]\!]
\end{array}
$$

The coding could be simplified by adding coactions, in the style of SA calculus of [LS00]. Coactions help serialize the steps of the protocol by means of the capability-cocapability synchronization. The lock ambient q would then be substituted by cocapability $\mathsf{coopen}\ p$, enforcing the opening of p only after the message has been sent.

Local and Self Invocation. Local method invocation within an ambient a is coded similarly to the previous case. Choosing $p, q \notin fn(M) \cup fn(P)$, one defines:

$$a \,\mathsf{local}\, \ell\langle M\rangle.P \triangleq (\nu p, q)\,(p[\![\,\mathsf{out}\ a.a\ \mathsf{send}\ \ell\langle M\rangle.\mathsf{in}\ a.q[\![\,\mathsf{out}\ p\,]\!]\,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P)$$

Relying upon this definition, it is then easy to define self-invocation within method bodies. To exemplify, consider the following process:

$$a[\![\,\ell_1(x) \triangleright \varsigma(z) z\ \mathsf{local}\ \ell_2\langle x\rangle :: \ell_2(x) \triangleright P\,;\, R\,]\!]$$

Invoking the method ℓ_1 from outside the object a results in the execution of the process P in parallel with R within a.

Child-to-parent. We conclude with a form of upward method invocation, whereby an object invokes a method provided by that object's parent. A first way of defining it might be:

$$a\ \mathsf{upsend}\ \ell\langle M\rangle.P \ \triangleq\ \mathsf{out}\ a.a\ \mathsf{send}\ \ell\langle M\rangle.\mathsf{in}\ a$$

But this is not fully satisfactory because requires a move of the sender. Alternatively, we can encode it by using some auxiliary ambient. Assume that the invocation occurs within an object b enclosed within a:

$$\begin{aligned}&a\ \mathsf{upsend}\ \ell\langle M\rangle.P \triangleq \\ &\quad (\nu\ p, q)\,(p[\![\,\mathsf{out}\ b.\mathsf{out}\ a.a\ \mathsf{send}\ \ell\langle M\rangle.\mathsf{in}\ a.\mathsf{in}\ b.q[\![\,\mathsf{out}\ p\,]\!]\,]\!] \mid \mathsf{open}\ q.\mathsf{open}\ p.P)\end{aligned}$$

To understand the definition, simply look at the chain of capabilities inside the ambient p, which corresponds to the steps in the protocol evolution. First, the ambient p leaves its parent ambients b, then a (that contains the method to be invoked), and performs the message send before being destroyed after the opening of the locking ambient q. One problem with the encoding is that it is context-dependent, since it uses the name b of the sender.

3.2 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. Let be $p, q \notin fn(P)$

$$\begin{aligned}!P \ \triangleq\ &(\boldsymbol{\nu} p, q)\ (p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P\ \mid \\ &\qquad p[\![\,! \triangleright \varsigma(z)(q[\![\,\mathsf{out}\ z.z\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P\,]\!])\,;\,]\!])\end{aligned}$$

The reduction for the encoding of $!P$ is then the following:

$$\begin{aligned}!P \triangleq\ &(\boldsymbol{\nu} p, q)\ \big(\underline{p\ \mathsf{downsend}\ !\langle\rangle}.\mathsf{open}\ q.P \mid p[\![\,! \triangleright \varsigma(z)(q[\![\cdots]\!])\,;\,]\!]\big)\\ \rightarrow\ &(\boldsymbol{\nu} p, q)\ \big(\mathsf{open}\ q.P \mid p[\![\,! \triangleright \varsigma(z)(\ldots)\,;\, q[\![\,\underline{\mathsf{out}\ p}.p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P\,]\!]\,]\!]\big)\\ \rightarrow\ &(\boldsymbol{\nu} p, q)\ \big(\underline{\mathsf{open}\ q}.P \mid q[\![\,p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P\,]\!] \mid p[\![\,! \triangleright \varsigma(z)(\ldots)\,;\,]\!]\big)\\ \rightarrow\ &(\boldsymbol{\nu} p, q)\ (P \mid p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P \mid p[\![\,! \triangleright \varsigma(z)(\ldots)\,;\,]\!])\\ \equiv\ &P \mid !P\end{aligned}$$

Notice that at each reduction step only one capability is ready to be exercised. Furthermore, the process P is activated only after the opening of the ambient q, hence it does not interfere with the protocol. We have then that the described protocol is a "precise" encoding of the replication (free from interferences).

3.3 Code on Demand

Even if we adopted RPC as primitive protocol for remote method invocation, the *Code on Demand* protocol used in [BC00,BCC00] is useful in several situation. The behavior of code on demand (cod) can be described as follows. A client c invokes a method ℓ on a server s; the server activates the method and then sends it back to the client for the latter to execute it. Formally this correspond to the following reduction rule:

$$
\begin{aligned}
&c[\![\,\mathrm{J}\,;\, s\ \mathsf{send\_cod}\ell\langle M\rangle.R \mid S\,]\!] \mid s[\![\,\mathrm{I} :: \ell(x) \triangleright \varsigma(z)Q\,;\,P\,]\!] \quad \rightarrow \\
&\qquad\qquad c[\![\,\mathrm{J}\,;\, Q\{z, x := s, M\} \mid R \mid S\,]\!] \mid s[\![\,\mathrm{I} :: \ell(x) \triangleright \varsigma(z)Q\,;\,P\,]\!]
\end{aligned}
$$

The protocol can be encoded by translating the caller and the called ambients as follows:

$$
\begin{aligned}
\mathit{server} &\triangleq s[\![\,\mathrm{I} :: \ell(u,v,x) \triangleright \varsigma(z)u[\![\,\mathsf{out}\ z.\mathsf{in}\ v.Q\,]\!]\,;\,P\,]\!] \\
\mathit{client} &\triangleq c[\![\,\mathrm{J}\,;\,(\boldsymbol{\nu} p)s\ \mathsf{send}\ \ell\langle p, c, M\rangle.\mathsf{open}\ p.R \mid S\,]\!]
\end{aligned}
$$

The protocol relies on the agreement between the server and the client upon the name of the ambient that carries the activated process back to the client. This name is decided locally by the client which passes it as an argument for the call together with its own name. Invoking $\ell\langle p, c.M\rangle$ spawns a new process on the server that simply carries the ambient p out of the server and back into the client c: once inside c, the transport ambient p is opened thus unleashing the process Q to be executed on the client.

Note that, if the second argument passed to the method ℓ were a path, rather than the client's name, then by slightly modifying the server we could have a more general protocol, where the client can choose where to receive and to execute the requested method (e.g. , in one of its subambients).

3.4 Updates

Following the standard definition of method override [AC96,FHM94] in formal calculi method updates for ambients can be formulated, informally, as follows: given the ambient $a[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\,Q\,]\!]$ we wish to replace the current definition P of ℓ by the new definition P' to form the ambient $a[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P'\,;\,Q\,]\!]$

Updates can be coded using a distinguished ambient as "updater". The updater carries the new method body and enters the updatable ambient a: 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, in an asynchronous setting: a similar encoding can be defined for synchronous updates. Moreover, we allow only local updates, that is, an ambient may only override methods contained in subambients (of course other kind of updates can be encoded, too)

Updates are denoted by a update $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$, read "the ℓ method at a gets definition P ". We define their behavior as follows: let first

$$a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \ \stackrel{\triangle}{=}\ \text{UPD}[\![\, \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ \mathsf{in}\ a\,]\!]$$

Then define an updatable ambient as follows

$$a^{\star}[\![\,\mathrm{I}\,;\,P\,]\!] \ \stackrel{\triangle}{=}\ a[\![\,\mathrm{I}\,;\,!(\mathsf{open}\ \text{UPD}) \mid P\,]\!]$$

Now, if we form the composition a update $\ell(\boldsymbol{x})\triangleright\varsigma(z)P' \mid a^{\star}[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\,Q\,]\!]$, the reduction for open enforces the expected behavior:

$$a\ \mathsf{update}\ \ell(\boldsymbol{x})\triangleright\varsigma(z)P' \mid a^{\star}[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\,Q\,]\!] \quad \rightarrow^{*} \quad a^{\star}[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P'\,;\,Q\,]\!]$$

Multiple updates for the same method may occur in parallel, in which case their relative order is established nondeterministically. The protocol, as defined, relies on the assumption that the name of the updater ambient carrying the new method body is globally known. A more realistic assumption is that updated ambient and the context agree on the name of the updater prior to start the protocol. This can be accomplished with a different definition of updatable ambient, one that assumes that updatable ambients come with an ad-hoc method that sets the appropriate conditions for the actual update to take place. The *upd* method below serves this purpose.

$$a^{\star}[\![\,\mathrm{I}\,;\,P\,]\!] \ \stackrel{\triangle}{=}\ a[\![\,\mathrm{I} :: \mathit{upd}(u) \triangleright \varsigma(z)\mathsf{open}\ u\,;\,P\,]\!]$$

Now, the updated protocol comprises two steps. First the updated ambient receives the name of the updater, and only then does the update take place:

$$a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \ \stackrel{\triangle}{=}\ (\boldsymbol{\nu}p)\,(a\ \mathsf{downsend}\ \mathit{upd}\langle p\rangle.p[\![\, \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ \mathsf{in}\ a\,]\!]\,)$$

3.5 Encoding the π-calculus

A final 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 [CG98] 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 an updatable ambient n, two locks n^i, and n^o and an auxiliary ambient $\bar{n}$ needed for the communication protocol based on RPC. The ambient n contains a method ch: a process willing to read from n installs itself as the body of this method, whereas a process willing to write on n invokes ch passing along the argument of the communication.

$$
\begin{aligned}
(ch\ n) &\triangleq n^{\star}[\![\, \mathtt{ch}(x) \triangleright \mathbf{0} \,]\!] \mid n^{i}[\![\ ]\!] \\
n!\langle y\rangle.Q &\triangleq \mathsf{open}\ n^{o}.n\ \mathsf{downsend}\ \mathtt{ch}(y).\mathsf{open}\ \bar{n}.(n^{i}[\![\ ]\!] \mid Q) \\
n?(x).P &\triangleq \mathsf{open}\ n^{i}.\ n\ \mathsf{update}\ \mathtt{ch}(x) \triangleright (\bar{n}[\![\mathsf{out}\ n.P]\!])\ .n^{o}[\![\ ]\!]
\end{aligned}
$$

The communication is then the following: a process $n!\langle y\rangle.Q$ writing y on n first attempts to grab the output lock n^o, then sends the message $\mathtt{ch}(y)$ to n, and after the end of the RPC protocol (i.e. after the opening of the carrier ambient $\bar{n}$), the process 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 ch 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 way of composing Q with the context:

$$
n!\langle y\rangle.Q \ \triangleq\ (\mathsf{open}\ n^{o}.n\ \mathsf{downsend}\ \mathtt{ch}(y).\mathsf{open}\ \bar{n}.(n^{i}[\![\ ]\!])) \mid Q
$$

Based on the this technique, we can encode the synchronous (and similarly, the asynchronous) polyadic π-calculus in ways similar to what is done in [CG99]. Each name n in the π-calculus becomes a quadruple of names in our calculus: the name n of the ambient dedicated to the communication, the names n^i and n^o of the two locks, and the name $\bar{n}$ of the auxiliary ambient. Therefore, communication of a π-calculus name becomes the communication of a quadruple of ambient names.

$$
\begin{aligned}
\langle\!\langle (\boldsymbol{\nu} n)P \rangle\!\rangle &\triangleq (\boldsymbol{\nu} n, \bar{n}, n^{i}, n^{o})(n^{i}[\![\ ]\!] \mid n^{\star}[\![\mathtt{ch}(x,\bar{x},x^{i},x^{o}) \triangleright \mathbf{0}]\!] \mid \langle\!\langle P \rangle\!\rangle) \quad \bar{n}, n^{i}, n^{o} \notin fn(\langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle n!\langle y\rangle.Q \rangle\!\rangle &\triangleq \mathsf{open}\ n^{o}.n\ \mathsf{downsend}\ \mathtt{ch}(y,\bar{y},y^{i},y^{o}).\mathsf{open}\ \bar{n}.(n^{i}[\![\ ]\!] \mid Q) \\
\langle\!\langle n?(x).P \rangle\!\rangle &\triangleq \mathsf{open}\ n^{i}.n\ \mathsf{update}\ \mathtt{ch}(x,\bar{x},x^{i},x^{o}) \triangleright (\bar{n}[\![\mathsf{out}\ n.P]\!])\ .n^{o}[\![\ ]\!] \\
\langle\!\langle P \mid Q \rangle\!\rangle &\triangleq \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\
\langle\!\langle !P \rangle\!\rangle &\triangleq !\langle\!\langle P \rangle\!\rangle \\
\langle\!\langle \mathbf{0} \rangle\!\rangle &\triangleq \mathbf{0}
\end{aligned}
$$

Fig. 6. Encoding of the synchronous π-calculus

The initialization of the ch 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.

A compositional encoding of the π-calculus channel-based communication in terms of message sends, can be defined in a way similar to that in [LS00], adding to the calculus coactions and relying on their ability to control/synchronize any computational step. See Section 5 for a more detailed discussion.

4 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 grammar productions for types are:

Signatures	$\Sigma ::=$	$(\ \ell_i(\mathscr{V}_i)\ )^{i \in I}$
Ambients	$\mathscr{A} ::=$	$\mathsf{Amb}[\Sigma]$
Capabilities	$\mathscr{C} ::=$	$\mathsf{Cap}[\Sigma]$
Processes	$\mathscr{P} ::=$	$\mathsf{Proc}[\Sigma]$
Values	$\mathscr{V} ::=$	$\mathscr{A} \mid \mathscr{C}$
Types	$\mathscr{T} ::=$	$X \mid \mathscr{A} \mid \mathscr{C} \mid \mathscr{P}$

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

The complete syntax of types contains type variables, that are used to deal with the dependency of method bodies on the *self* parameter. In fact, due to ambient opening, a method residing in an ambient a may be reinstalled inside a new ambient that opens a and that may have a richer interface; thus the type of the *self* variable may be dynamically rebound to a different ambient type. As a consequence, to ensure sound typings of method invocations, method bodies are typed in a context that assumes the so-called *MyType* [Bru94] typing for the *self* variable, i.e. a match-bounded type variable representing the type of all ambients where the method can be reinstalled, via opening. In particular, we use a restricted form of *matching* relation [Bru94], where a type variable X, representing a *self* type, may appear in the context only match-bounded by an ambient type (i.e. $X \mathrel{<\!\#} \mathscr{A}$). Furthermore, we syntactically restrict our signatures, and consequently our ambient, capability and process types, to not contain type variables. As a consequence, the type system does not support *MyType* method specialization [Bru94,FHM94], the OO-typing technique that allows methods's types to be specialized when they are inherited (or, in our context, when they are subsumed in an opening ambient). Instead, in our calculus a method body has always the same type (that is, the one declared in Σ), independently of the dynamic binding of its *self* variable. This is not surprising, as our method bodies are processes with no return value, hence they are dealt with essentially as methods with return type `unit` in imperative object calculi.

4.1 Type System

The typed syntax of the calculus is described by the productions in Figure 7 :

Interfaces	I	$::= \ell(x) \triangleright \varsigma(z)P \mid \mathrm{I} :: \mathrm{I} \mid \varepsilon$
Processes	P	$::= \mathbf{0} \mid P \vert P \mid a[\![\mathrm{I};\, P]\!] \mid (\boldsymbol{\nu} x{:}\mathscr{A})P \mid M.P$
Expressions	M	$::= x \mid (M_1, \ldots, M_n) \mid x \;\mathsf{send}\; \ell\langle M\rangle \mid \mathsf{in}\; x \mid \mathsf{out}\; x \mid \mathsf{open}\; x \mid \varepsilon$

Fig. 7. Typed syntax for ambients

As we said in Section 2, we take method names to be fixed *labels* that may not be passed as values, nor restricted. The first restriction is justified by the fact that method names are part of the structure of ambient (capability and process) types; as a consequence, lifting this restriction would be possible but it would make our types (first-order) dependent types. Instead, lifting the second restriction is possible, and in fact not difficult, even though it complicates the format of the typing rules . For this reason we will disregard this issue in what follows.

The structure of contexts and judgments is defined by the productions below, where we assume $\mathscr{W}$ to range over the set $\{X, \mathscr{A}, \mathscr{C}\}$ of extended value types:

Contexts	$\Gamma ::= \varnothing \mid \Gamma, x : \mathscr{W} \mid \Gamma, X \lessdot\!\!\# \mathscr{A}$
Judgements J	$::= \Gamma \vdash M : \mathscr{W} \mid \Gamma \vdash X \lessdot\!\!\# \mathscr{A} \mid \Gamma \vdash P : \mathscr{P} \mid \Gamma \vdash \mathscr{T} \mid \Gamma \vdash \diamond$

Fig. 8. Contexts and typing judgments

The complete set of typing rules is presented in Appendix A; below, we discuss the most interesting ones.

Method signatures, associated with ambient types, are traced by the types Cap, of capabilities, to allow an adequate typing of messages, mobility and opening.

$$\text{(OPEN)}\quad \frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma]}{\Gamma \vdash \mathsf{open}\; a : \mathsf{Cap}[\Sigma]}$$

The rule (OPEN) for opening an ambient requires precise knowledge of the type of the ambient being opened: consequently, the type of the ambient must be an ambient type, not a type variable. An opening is now legal under the condition that the signature of the opening ambient be equal to (in fact, contain, 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 their original typing must be preserved.

$$\text{(MESSAGE)}\quad \frac{\Gamma \vdash a : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \mathrel{\lessdot\!\#} \mathsf{Amb}[\ \ell(\mathscr{V}')\ ] \quad \Gamma \vdash M' : \mathscr{V}'}{\Gamma \vdash a\ \mathsf{send}\ \ell\langle M' \rangle : \mathsf{Cap}[\Sigma]}$$

Rule (MESSAGE) says that an invocation for method ℓ on an expression a requires the type of a to match an ambient type containing the method ℓ. Note that the type of a may either be an ambient type matching (i.e. "longer" then) $\mathsf{Amb}[\ell(\mathscr{V}')]$, or else an unknown type (i.e. a type variable) occurring match-bounded in the context Γ. Since the body of the invoked method is not executed in the same ambient that contains the send capability (due to the RPC semantics), no constraint is imposed on the type of the send capability. Of course, in order for the expression to type check, the message argument and the method parameters must have the same type. [4]

$$\text{(AMB)}\quad (\Sigma = (\ \ell_i(\mathscr{V}_i)\ )^{i \in I})\quad \frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma] \quad \Gamma, Z \mathrel{\lessdot\!\#} \mathsf{Amb}[\Sigma], z{:}Z, x_i{:}\mathscr{V}_i \vdash P_i : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash a[(\ell_i(x_i) \triangleright \varsigma(z)P_i)\ ^{i \in I}\ ; P] : \mathsf{Proc}[\Sigma']}$$

Rule (AMB) types ambients similarly to objects in the object calculi of [AC96]: each method is typed under the assumptions that (i) the self parameter has a type that matches the type of the enclosing ambient, (ii) method parameters have the declared type, and (iii) method bodies must be typable with a process type that agrees with the type of the enclosing ambient and that is independent on the type of *self* (i.e. disallowing *MyType* method specialization). Moreover, the rule requires the local process to have a process type that agrees with the type of the enclosing ambient. 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).

Note that the match-binding for the type of the *self* variable ensures that methods local to ambient a are well-typed also within any other ambient that might eventually open a. Also the rule requires exact knowledge of the true type of the ambient's name; a structural rule allowing the name of the ambient to be typed with a match-bounded type variable would break type soundness, since we would not have a precise control of the openings of that ambient (see rule (OPEN)).

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

(Match Amb)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i\in 1..n+k}] \Lleftarrow\!\!\#\, \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i\in 1..n}]}$$

(Sub Cap)

$$\frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \leq \mathsf{Cap}[\Sigma']}$$

(Sub Proc)

$$\frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \leq \mathsf{Proc}[\Sigma']}$$

Non-trivial subtyping is defined for capability and process types: specifically, a capability (resp. process) type $\mathsf{Cap}[\Sigma]$ (resp. $\mathsf{Proc}[\Sigma]$) is a subtype of any capability (resp. process) type whose associated signature (set theoretically) contains Σ. The resulting notion of subtyping is reminiscent of the notion of subtyping in *width* distinctive of type systems for object and record calculi. Width subtyping must be disallowed over ambient types to ensure sound uses of the `open` capability: intuitively, when opening an enclosed ambient, one again needs 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. Nevertheless, we have matching relation between ambient types, that ensures sound typing of methods even when they are merged, via opening, in a "larger" ambient.

The complete set of subtyping and matching rules includes the standard rules for reflexivity and transitivity (not shown). Also, as customary, the subtyping relation is endowed in the type system via a subsumption rule.

4.2 Subject Reduction and Type Soundness

We conclude the description of the basic type system with a proof of subject reduction. The proof is rather standard, and only sketched due to lack of space.

Lemma 1 (Substitution).

1. *If* $\Gamma, x{:}\mathscr{W} \vdash P : \mathscr{P}$ *and* $\Gamma \vdash M : \mathscr{W}$*, then* $\Gamma \vdash P\{x{:}= M\} : \mathscr{P}$.
2. *If* $\Gamma, Z \Lleftarrow\!\!\#\, \mathscr{A}, z{:}Z \vdash P{:}\mathscr{P}$ *and* $\Gamma \vdash a{:}\mathscr{A}'$, $\Gamma \vdash \mathscr{A}' \Lleftarrow\!\!\#\, \mathscr{A}$, *then* $\Gamma \vdash P\{z{:}= a\}{:}\mathscr{P}$.

Proof. By induction on the derivation of the first judgment in hypothesis.

Proposition 1 (Subject Congruence).

1. *If* $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.
2. *If* $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $Q \equiv P$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.

Proof. By simultaneous induction on the derivations of $P \equiv Q$ *and* $Q \equiv P$.

Lemma 2 (Bounded Weakening).

1. *If* $\Gamma, x : \mathscr{W} \vdash P : \mathscr{P}$ *and* $\Gamma \vdash \mathscr{W}' \leq \mathscr{W}$ *then* $\Gamma, x : \mathscr{W}' \vdash P : \mathscr{P}$.
2. *If* $\Gamma, Z \Lleftarrow\!\!\#\, \mathscr{A}, z{:}Z \vdash P{:}\mathscr{P}$ *and* $\Gamma \vdash \mathscr{A}' \Lleftarrow\!\!\#\, \mathscr{A}$ *then* $\Gamma, Z \Lleftarrow\!\!\#\, \mathscr{A}', z{:}Z \vdash P{:}\mathscr{P}$.

Proof. *By induction on the derivation of the first judgment in hypothesis.*

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

Proof. *By induction on the derivation of* $P \rightarrow Q$*, and a case analysis on the last applied rule.*

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 statically detect are those of the kind "message not understood" which are distinctive of object calculi. With the current definition of the reduction relation such errors may 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 it

$$a[\![\mathrm{I}\,;\, P \mid b\ \mathsf{send}\ \ell\langle M\rangle.Q\,]\!] \mid b[\![\,\mathrm{J}\,;\, R\,]\!] \;\rightarrow\; a[\![\mathrm{I}\,;\, P \mid \mathtt{ERR}\,]\!] \mid b[\![\,\mathrm{J}\,;\, R\,]\!] \qquad (\ell \notin \mathrm{J})$$

together with the rules that propagate errors in every context. 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 rule above is meaningful also in the presence of multiple ambients with equal name, as our type system (like those of [CG99,CGG99,LS00]) ensures that ambients with the same name have also the same type. Therefore if a message ℓ is absent from a given ambient b, it will also be absent from all ambients named b.

If we assume that ERR is 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 (Soundness). *For every* Γ*,* P*, if* $\Gamma \vdash P : T$*, then* $P \not\rightarrow^* \mathtt{ERR}$.

5 Adding coaction: SA^{++}

In [LS00], Levi and Sangiorgi show that the calculus of Mobile Ambients can be refined in order to have a richer algebraic theory and prove useful properties. To that end, they define the *Safe Ambients* calculus, where each MA's capability is combined with a dual cocapability, and where a computational reduction step is the result of a capability-cocapability synchronization. Thus an interaction between two ambients only happens when both ambients agree on their intentions.

Following their work it is not difficult to add cocapabilities to MA^{++} calculus we presented here, obtaining what we call SA^{++} . In particular, the SA^{++} calculus contains a cocapability $\mathsf{listen}\ a$, that is the dual of the capability $a\ \mathsf{send}$, and whose meaning is that the ambient a is ready to serve an invocation to one of its methods.

For reasons of space, we do not describe this extension in full details, but we want nevertheless to point out its advantages by showing how it allow us to derive a simpler and compositional encoding of the π-calculus.

$$
\begin{aligned}
\langle\!\langle n?(\boldsymbol{x}).P \rangle\!\rangle &\triangleq (\boldsymbol{\nu} p)(n[\, ch(\boldsymbol{x}) \triangleright p[\,\mathsf{out}\ n.\mathsf{coopen}\ p.\langle\!\langle P \rangle\!\rangle\,]\ ;\ \mathsf{listen}\ n.\mathsf{coout}\ n\,]\ \mid \mathsf{open}\ p) \\
\langle\!\langle n!\langle \boldsymbol{x} \rangle \rangle\!\rangle &\triangleq n\ \mathsf{downsend}\ ch\langle \boldsymbol{x} \rangle \\
\langle\!\langle (\boldsymbol{\nu x})P \rangle\!\rangle &\triangleq (\boldsymbol{\nu x})\langle\!\langle P \rangle\!\rangle \\
\langle\!\langle P \mid Q \rangle\!\rangle &\triangleq \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\
\langle\!\langle !P \rangle\!\rangle &\triangleq\ !\langle\!\langle P \rangle\!\rangle \\
\langle\!\langle \mathbf{0} \rangle\!\rangle &\triangleq \mathbf{0} \\
\langle\!\langle n \rangle\!\rangle &\triangleq n \\
\langle\!\langle Ch(\boldsymbol{T}) \rangle\!\rangle &\triangleq \mathsf{Amb}[ch(\langle\!\langle \boldsymbol{T} \rangle\!\rangle)]
\end{aligned}
$$

Fig. 9. Encoding of the asynchronous π-calculus

Every input on a channel n generates a new ambient named n, waiting to synchronize with an output on n. Having received input, the transport ambient p carries (the encoding of) P out of n. Once outside n, p is dissolved thus unleashing the continuation process P. It is instructive to notice that the ambient n is left without capabilities after having let the transport p out. As such, after synchronization, n is unavailable for interactions with the context, and thus behaviorally equivalent to the null process and garbage collectable.

Note also that, dealing only with the processes yielding from the encoding of π-calculi processes, the parent-to-child invocation protocol is guaranteed to be executed without interferences.

6 Related work

In the literature on concurrent object oriented programming, papers can be distinguished in two basic categories. The first class contains works that provide semantics to objects by encoding them into process calculi. Works in the second class study calculi where primitives for objects and for concurrent processes coexist.

Systematic translations of objects into π-calculus can be found, for instance, in [Wal95,HK96,San98,KS98]. Works that belong to the second approach are much closer to what we do here. Among these it is worth to mention the approaches of [Vas94,PT95,FMLR00] which, given a name-passing calculus, build high-level constructors distinctive of object-oriented languages. A complementary approach is followed by [GH98] and [DBF96] since they add primitives for concurrency to the imperative object-oriented calculus of, respectively, [AC96] and [FHM94]. Aspects of distribution are taken into account in [NHKM99,Jef00].

We present next a detailed discussion on works most related to our.

The concς calculus. In [GH98] the authors present a concurrent object calculus (concς) that consists of Abadi and Cardelli's imperative object calculus extended with primitives for parallel composition, restriction and synchronization via mutexes. They also show that existing type systems for the underlying object calculus can be smoothly and soundly extended to accommodate concurrency.

The basic difference between this work and that we presented, is the fact that [GH98] does not deal with process mobility. In [GH98] distribution aspects are absent, while in our framework objects may move through a hierarchy of nested locations, and communication (method invocation) often requires mobility. Moreover, in our framework, due to the interplay between the dynamic nesting structure and the communication primitives, more method invocation styles can be modeled.

On the other hand, the semantics of method invocation in [GH98], as well as in our work, is based on the idea of self-substitution distinctive of [AC96]. As in the work presented here, in [GH98] objects are explicitly named, thus what gets substituted for the *self* variable is the name of the object rather then the object itself.

A distinctive feature of [GH98] is the fact that the syntax of concς includes sequential composition of expressions that return results. This contrasts with what happens in most formalisms based on processes ([Vas94,PT95,Wal95,KS98]), where the operation of returning a result is translated into sending a message on a result channel. Even though we did not explicitly address the problem of returning a result, it is easy to extend our framework by endowing agent interfaces not only with methods, but also with *fields* whose invocation returns an expression.

A distributed version of concς is studied in [Jef00], where the syntax of the calculus is enriched with a notion of *location*, and threads are allowed to migrate between locations. Contrary to our framework, in [Jef00] the author considers a very simple, fixed flat set of locations, with no routing information, no dynamic location creation or hierarchy of locations. Moreover, in [Jef00] only a subset of objects (*serializable* objects) can be sent across the network, and only the so-called *located objects* can be accessed via remote threads.

The Ojeblik calculus. In [NHKM99] authors present Ojeblik, an object-based language that represents the concurrent core of Obliq ([Car95]), Cardelli's lexically scoped distributed programming language. In this setting, mobility of objects is rendered via a migration mechanism that is carried out by creating a copy of the object at the target site and then modifying the original (local) object such that it forwards future requests to the new (remote) object. Moreover, lexical scoping of Obliq permits to safely ignore aspects of distribution. Migration is then *correct* if the behavior of an object is transparent to whether the object has migrated or not.

Our approach is very different since, in a way similar to that of Ambient Calculus, we assume that the process $a[\mathrm{I}\,;P]$ is an abstraction for both an agent (client) and an object (server). This implies that in our framework mobile objects move without the burden of future obligations at the source location. A client agent willing to invoke a method of a server object, in turn, must approach the server in order to start the communication protocol.

In addition, while the work on Ojeblik does not address typing issues, we developed a rich type theory showing how advances in type system for object-oriented languages can be reused in the context of calculi of mobile agents.

7 Future Work

MA^{++} is a core calculus on the top of which many other extensions, besides the one with coactions and single threaded types [LS00] can be defined.

A first example is the addition of fields. Unlike what happens in object calculi, where fields can be obtained as parameter-less methods, here fields cannot be encoded. Calling a method does not return a value, but instead spawns a process. The solution is to explicitly add new syntax for fields, which operationally instead of triggering a process returns terms.

A different possibility is to extend the calculus so that method names have not a distinguished status but are dealt with as ordinary names. This would allow one to restrict them, thus obtaining private methods, and to communicate them, thus obtaining dynamic messages. This is a straightforward modification in the untyped calculus but it is quite problematic in the typed case since the possibility of communicating method names would naturally give rise to dependent types.

Finally we could imagine to define security policies for MA^{++} and try to apply it to specify and verify real case examples.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[BC00] M. Bugliesi and G. Castagna. Mobile objects. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. Electronic Proceedings.

[BCC00] M. Bugliesi, G. Castagna, and S. Crafa. Typed mobile objects. In *Proceedings of CONCUR 2000 (11th. International Conference on Concurrency Theory)*, number 1877 in Lecture Notes in Computer Science, pages 504–520. Springer, 2000.

[Bru94] B. Bruce, K. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 1(4):127–206, 1994.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Car99] L. Cardelli. Abstractions for mobile computations. In *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science, pages 51–94. Springer, 1999.

[CG98] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL'98*. ACM Press, 1998.

[CG99] L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM Press, 1999.

[CGG99] 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.

[DBF96] P Di Blasio and K. Fisher. A calculus for concurrent objects. In *CONCUR '96*, number 1119 in Lecture Notes in Computer Science, pages 655–670. Springer, 1996.

[FHM94] 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.

[FMLR00] Cédric Fournet, Luc Maranget, Cosimo Laneve, and Didier Rémy. Inheritance in the Join Calculus. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*. Springer, December 2000.

[GH98] 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.

[HK96] H. Huttel and J. Kleist. Objects as mobile processes. Technical Report Research Series RS-96-38, BRICS, 1996. Presented at MFPS '96.

[Jef00] A. Jeffrey. A distributed object calculus. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. Electronic Proceedings.

[KS98] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *PROCOMET '98 (IFIP Working Conference on Programming Concepts and Methods)*. North-Holland, 1998.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL '00*, pages 352–364. ACM Press, 2000.

[NHKM99] U Nestmann, H. Huttel, J. Kleist, and M. Merro. Aliasing models for object migration. In *Proceedings of Euro-Par'99*, number 1685 in Lecture Notes in Computer Science, pages 1353–1368. Springer, 1999.

[PT95] 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, April 1995.

[San98] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *IC*, 143(1):34–73, 1998.

[Vas94] 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.

[Wal95] D.J Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

A Typing rules

Context formation

$$\text{(ENV-EMPTY)}\;\frac{}{\varnothing \vdash \diamond} \qquad \text{(ENV-}x\text{)}\;\frac{\Gamma \vdash \mathscr{W} \quad x \notin Dom(\Gamma)}{\Gamma, x : \mathscr{W} \vdash \diamond} \qquad \text{(ENV-}X\text{)}\;\frac{\Gamma \vdash \diamond \quad X \notin Dom(\Gamma)}{\Gamma, X \triangleleft\!\# \mathscr{A} \vdash \diamond}$$

Type formation

$$\text{(TYPE X)}\;\frac{\Gamma, X \triangleleft\!\# \mathscr{A}, \Gamma' \vdash \diamond}{\Gamma, X \triangleleft\!\# \mathscr{A}, \Gamma' \vdash X} \qquad \text{(TYPE AMB)}\;\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[\Sigma]} \qquad \text{(TYPE CAP)}\;\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Cap}[\Sigma]} \qquad \text{(TYPE PROC)}\;\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Proc}[\Sigma]}$$

Matching : Reflexivity, Transitivity and the following

$$\text{(Match X)}\quad \frac{\Gamma, X \Lleftarrow\!\!\# \mathscr{A}, \Gamma' \vdash \diamond}{\Gamma, X \Lleftarrow\!\!\# \mathscr{A}, \Gamma' \vdash X \Lleftarrow\!\!\# \mathscr{A}} \qquad \text{(Match Amb)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i \in 1..n+k}] \Lleftarrow\!\!\# \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i \in 1..n}]}$$

Subtyping and subsumption : Reflexivity, Transitivity and the following

$$\text{(Sub Cap)}\quad \frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \le \mathsf{Cap}[\Sigma']} \qquad \text{(Sub Proc)}\quad \frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \le \mathsf{Proc}[\Sigma']} \qquad \text{(Subsumption)}\quad \frac{\Gamma \vdash A : \mathscr{T} \quad \mathscr{T} \le \mathscr{T}'}{\Gamma \vdash A : \mathscr{T}'}$$

Expressions

$$\text{(name/var)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)} \qquad (\varepsilon)\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \varepsilon : \mathsf{Cap}[\Sigma]} \qquad \text{(path)}\quad \frac{\Gamma \vdash M_1 : \mathsf{Cap}[\Sigma] \quad \Gamma \vdash M_2 : \mathsf{Cap}[\Sigma]}{\Gamma \vdash M_1.M_2 : \mathsf{Cap}[\Sigma]}$$

$$\text{(open)}\quad \frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma]}{\Gamma \vdash \mathsf{open}\ a : \mathsf{Cap}[\Sigma]} \qquad \text{(inout)}\quad \frac{\Gamma \vdash M : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \Lleftarrow\!\!\# \mathsf{Amb}[\Sigma] \quad (M' \in \{\mathsf{in}\ M, \mathsf{out}\ M\})}{\Gamma \vdash M' : \mathsf{Cap}[\Sigma']}$$

$$\text{(Message)}\quad \frac{\Gamma \vdash a : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \Lleftarrow\!\!\# \mathsf{Amb}[\ \ell(\mathscr{V}')\ ] \quad \Gamma \vdash M' : \mathscr{V}'}{\Gamma \vdash a\ \mathsf{send}\ \ell\langle M' \rangle : \mathsf{Cap}[\Sigma]}$$

Processes

$$\text{(pref)}\quad \frac{\Gamma \vdash M : \mathsf{Cap}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash M.P : \mathsf{Proc}[\Sigma]} \qquad \text{(par)}\quad \frac{\Gamma \vdash P : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash Q : \mathsf{Proc}[\Sigma]}{\Gamma \vdash P \mid Q : \mathsf{Proc}[\Sigma]}$$

$$\text{(restr)}\quad \frac{\Gamma, x{:}\mathscr{A} \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash (\boldsymbol{\nu} x{:}\mathscr{A})P : \mathsf{Proc}[\Sigma]} \qquad \text{(dead)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \mathsf{Proc}[\Sigma]}$$

$$\text{(Amb)} \qquad (\Sigma = (\ \ell_i(\mathscr{V}_i)\ )^{i \in I})$$
$$\frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma] \quad \Gamma, Z \Lleftarrow\!\!\# \mathsf{Amb}[\Sigma], z{:}Z, x_i{:}\mathscr{V}_i \vdash P_i : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash a[(\ell_i(x_i) \triangleright \varsigma(z)P_i)\ ^{i \in I}\ ;\ P] : \mathsf{Proc}[\Sigma']}$$