The M-calculus: A Higher-Order Distributed Process Calculus

Alan Schmitt
INRIA
alan.schmitt@inria.fr

Jean-Bernard Stefani
INRIA
jean-bernard.stefani@inria.fr

ABSTRACT

This paper presents a new distributed process calculus, called the M-calculus, that can be understood as a higher-order version of the Distributed Join calculus with programmable localities. The calculus retains the implementable character of the Distributed Join calculus while overcoming several important limitations: insufficient control over communication and mobility, absence of dynamic binding, and limited locality semantics. The calculus is equipped with a polymorphic type system that guarantees the unicity of locality names, even in presence of higher-order communications – a crucial property for the determinacy of message routing in the calculus.

Categories and Subject Descriptors

F.m [**Theory of Computation**]: Miscellaneous

General Terms

Design, Languages, Theory

Keywords

Process Calculi, Distribution, Mobility, Higher Order, Type Systems

1. INTRODUCTION

Among the process calculi which have been introduced over the past decade to serve as a basis for a distributed and mobile programming model, the Distributed Join calculus [7, 6, 11] constitutes an interesting milestone. It provides a distributed programming model with hierarchical fail-stop localities, transparent mobility and communications, and it can be efficiently implemented. The Distributed Join calculus, however, has several limitations:

- It offers insufficient control over communication and process mobility, which is an issue in distributed environments where security is a primary concern. For instance, it is not possible to prevent a locality from migrating to another locality, except by forcing its failure. Also, once a resource (a Distributed Join calculus definition) has been defined and communicated, it is very difficult to prevent access to that resource or to define the equivalent of a firewall [4].
- It does not support dynamic binding. In a distributed programming model, it is important to provide both local and remote equivalent of libraries or services, because of the cost, safety, and security considerations that may apply. Thus, it should be possible to access identically named libraries or services (like a print service) at different sites. In the Distributed Join calculus such a choice is not directly available since each definition is uniquely defined: every resource is permanently bound to a single locality.
- It does not support the definition of localities with different semantics. For instance, localities in the Distributed Join calculus are defined to be fail-stop. While it would be possible to change the semantics of the calculus to accommodate different failure modes (such as omission or byzantine failures), the question remains as to how one can combine different failure modes within the same calculus. Likewise, one could require a locality to be endowed with a particular form of access control (e.g. restricting access to resources within a locality to principals appearing in an access control list).

The M-calculus presented in this paper is designed to overcome the limitations of the Distributed Join calculus while preserving some of its key features, notably, its concept of hierarchical localities (which is crucial to deal with security, migration, and failures), its notion of multiway synchronization, and its implementable character.

Specifically, the main contributions of this paper are:

- the notion of *programmable locality*, that generalizes the different concepts of locality found in the Distributed Join calculus and other distributed process calculi;
- the conjunction of higher-order processes and hierarchical programmable localities to unify communication and process migration, combining the possibility of

This work has been supported in part by the Mikado IST Global Computing Project (IST-2001-32222)

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

P	$::=$		process
		$\mathbf{0}$	null process
	$\mid$	V	value
	$\mid$	$\mathbf{a}(P)[P]$	locality
	$\mid$	$P \mid P$	parallel composition
	$\mid$	$P P$	application
	$\mid$	$(P, \ldots, P)$	tuple
	$\mid$	$\nu n.P$	restriction
	$\mid$	$([\mu = V]P, P)$	conditional
	$\mid$	$\langle J \triangleright P \rangle$	reaction rule
	$\mid$	$\mathtt{pass}\ V$	passivation
V	$::=$		value
		$()$	null value
	$\mid$	u	name
	$\mid$	$(V, \ldots, V)$	tuple
	$\mid$	$\lambda x.P$	abstraction
J	$::=$		Join pattern
		$\mathbf{r}\widetilde{x}$	message
	$\mid$	$J \mid J$	synchronization

Figure 1: Syntax of the M-calculus

n	$::=$		resolved name
		r	resource name
	$\mid$	a	locality name
$\mathbf{r}$	$::=$		variable resource name
		r	resource name
	$\mid$	x	variable
$\mathbf{a}$	$::=$		variable locality name
		a	locality name
	$\mid$	x	variable
u	$::=$		name
		$\mathbf{a}$	variable locality name
	$\mid$	$\mathbf{r}$	variable resource name
	$\mid$	$\mathbf{a.r}$	located resource
μ	$::=$		name pattern
		n	resolved name
	$\mid$	$\_$	wildcard

Figure 2: Names

$$
\begin{array}{rcl}
\mathbf{E} & ::= & (\cdot) \mid \mathbf{E}V \mid P\mathbf{E} \mid \nu n.\mathbf{E} \mid (\mathbf{E} \mid P) \mid (P \mid \mathbf{E}) \\
& \mid & a(P)[\mathbf{E}] \mid a(\mathbf{E})[P] \mid (P_1, \ldots, \mathbf{E}, \ldots, P_n)
\end{array}
$$

Figure 3: Evaluation Contexts

transparent routing as in the Distributed Join calculus with fine-grained ambient-like control over information exchange;

- the introduction of a passivation operator as a key primitive for programming different forms of control that can be exercised by localities;
- the definition of a type system that guarantees the determinacy of the routing mechanism by ensuring that every locality bears a unique name, even in presence of higher-order communication.

A programmable locality in the M-calculus (or locality, in brief) has a name and contains two processes: a *controller*, which filters incoming and outgoing messages, and a *content*. In order to apply the same control mechanisms to remote communication and process migration, the latter is just communication of a *thunk*, i.e. a frozen process. A running locality may be frozen by its controller using the *passivation* operator. This operator takes a function that defines the operations to apply on the controller and content processes of a locality, such as sending them in a remote message, discarding them, or modifying them.

The paper is organized as follows. Section 2 defines the syntax and operational semantics of the M-calculus. Section 3 introduces a type system that ensures the unicity of locality names. Section 4 gives an encoding of the Join calculus that illustrate the versatility of the calculus. Section 5 discusses related work. Section 6 concludes the paper.

2. THE M-CALCULUS

We present in this section the M-calculus, introducing the syntax and the semantics as we describe local communication, remote communication, control, and migration. The syntax is summarized in figures 1 and 2. In the following, we use bold fonts for identifiers that may stand for names or variables, as described in figure 2.

Communication takes the form of an asynchronous, channel-based, point-to-point exchange of messages, reflecting the dominant mode of communication in current large scale networks. Channels are called *resources* and we assume there is an infinite countable set of resource names. We let r range over this set. A local message is an application of a resource name to a tuple of values $r\widetilde{V}$. Receivers in the M-calculus are reaction rules composed of a multiway synchronization pattern (similar to the one proposed informally by Milner for his "Polynomial π-calculus" and to Join patterns), and of a guarded process. Every reaction rule $\langle r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n} \triangleright P \rangle$ *defines* the resource names $r_1, \ldots, r_n$. The formal parameters $\widetilde{x_1}, \ldots, \widetilde{x_n}$ are tuples of *variables*, and we assume there is an infinite countable set of variables. The local communication rule is very similar to the JOIN rule of the Join calculus, substituting message arguments for formal arguments in the guarded process (rule R.RES of figure 4). We remark that the reaction rule is replicated: it does not disappear after reduction. We also remark that a reaction rule does not bind its defined names. New resource names are introduced and bound using a restriction operator $\nu r.P$.

In the reduction rules of figure 4, rule R.CONTEXT uses evaluation contexts defined in figure 3, and rule R.EQUIV uses structural equivalence. Structural equivalence, $\equiv$, is the smallest equivalence relation that satisfies the rules given in Figure 5, where the parallel composition operator $\mid$ for processes is taken to be commutative and associative, with $\mathbf{0}$ as its neutral element. The structural rules comprise scope extrusion rules for the restriction operator, standard rules for equivalence under α-conversion, and congruence for evaluation contexts. Equivalence of two processes P and Q up to α-conversion is noted $P =_\alpha Q$. We recall that in $\nu n.P$, $\lambda x.P$, and $\langle \mathbf{r}_1\widetilde{x_1} \mid \ldots \mid \mathbf{r}_n\widetilde{x_n} \triangleright P \rangle$, the names and variables n, x, and $\widetilde{x}_i$ are bound in P. Free names of a process P are defined as usual and written $fn(P)$. We recall that the defined names of a reaction rule $\langle J \triangleright P \rangle$ are free.

$$\frac{}{(\lambda x.P)V \rightarrow P\{^{V}/_{x}\}} \text{ [R.BETA]}$$

$$\frac{match(\mu, V)}{([\mu = V]P, Q) \rightarrow P} \text{ [R.IF.THEN]}$$

$$\frac{\neg match(\mu, V)}{([\mu = V]P, Q) \rightarrow Q} \text{ [R.IF.ELSE]}$$

$$\frac{}{a(\mathsf{pass}\ V \mid P)[Q] \rightarrow V a(\lambda.P)(\lambda.Q)} \text{ [R.PASSIV]}$$

$$\frac{\langle J \triangleright P \rangle = \langle r_1 \widetilde{x_1} \mid \ldots \mid r_n \widetilde{x_n} \triangleright P \rangle}{\langle J \triangleright P \rangle \mid r_1 \widetilde{V_1} \mid \ldots \mid r_n \widetilde{V_n} \rightarrow \langle J \triangleright P \rangle \mid P\{^{\widetilde{V_i}}/_{\widetilde{x_i}}\}} \text{ [R.RES]}$$

$$\frac{P \rightarrow Q}{\mathbf{E}(P) \rightarrow \mathbf{E}(Q)} \text{ [R.CONTEXT]}$$

$$\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q} \text{ [R.EQUIV]}$$

Figure 4: Computing Rules

$$\frac{n \notin fn(Q)}{(\nu n.P) \mid Q \equiv \nu n.P \mid Q} \text{ [S.NU.PAR]}$$

$$\frac{n \notin fn(Q)\ \wedge\ n \neq a}{a(\nu n.P)[Q] \equiv \nu n.a(P)[Q]} \text{ [S.NU.CTRL]}$$

$$\frac{n \notin fn(P)\ \wedge\ n \neq a}{a(P)[\nu n.Q] \equiv \nu n.a(P)[Q]} \text{ [S.NU.CONT]} \qquad \frac{P =_\alpha Q}{P \equiv Q} \text{ [S.}\alpha\text{]}$$

$$\frac{P \equiv Q}{\mathbf{E}(P) \equiv \mathbf{E}(Q)} \text{ [S.CONTEXT]}$$

Figure 5: Structural Equivalence

As a communication example, we may write a reference cell process as in the Join calculus:

$$\left(\begin{array}{ll} \nu s. & \langle get(k) \mid s(st) \triangleright k(st) \mid s(st) \rangle \\ \mid & \langle set(st') \mid s(st) \triangleright s(st') \rangle \\ \mid & s(0) \end{array} \right) \mid get(print) \mid set(3)$$

which may reduce to $print(0)$ or $print(3)$.

We now describe remote communication of asynchronous messages. In many calculi, remote communication involves two steps: resolving where to send the message, and sending it. For instance, in the Distributed Join calculus, every channel is defined in at most one location and definitions cannot move from one location to another. Thus a defined channel name is unambiguously associated to the location containing its definition. In the dynamic Join calculus [20], the destination for a dynamic message is resolved according to the channel name and the current position of the message. In the Ambient calculus, an ambient migrates according to the explicit capabilities that it expresses and its local environment. The destination of a message in the Box-π calculus [21] also depends on the immediate environment of the message. In order to avoid restricting the calculus to one particular semantics, we let the resolving step be a part of the calculus: a remote message has the form $a.r\widetilde{V}$, where a is the explicit destination of the message, which

$$\frac{}{a(P \mid a.r\widetilde{V})[Q] \rightarrow a(P \mid r\widetilde{V})[Q]} \text{ [R.A.CTRL.FINAL]}$$

$$\frac{}{a(P)[Q \mid a.r\widetilde{V}] \rightarrow a(P)[Q \mid r\widetilde{V}]} \text{ [R.A.CONT.FINAL]}$$

$$\frac{b \in locs(P) \cup locs(Q) \cup \{a\}}{b.r\widetilde{V} \mid a(P)[Q] \rightarrow a(P \mid \mathsf{i}(b, r, \widetilde{V}))[Q]} \text{ [R.A.IN]}$$

$$\frac{b \in locs(Q) \setminus locs(P) \qquad b \neq a}{a(P \mid b.r\widetilde{V})[Q] \rightarrow a(P)[Q \mid b.r\widetilde{V}]} \text{ [R.A.CTRL.TO.CONT]}$$

$$\frac{b \notin locs(P) \cup locs(Q) \qquad b \neq a}{a(P \mid b.r\widetilde{V})[Q] \rightarrow a(P)[Q] \mid b.r\widetilde{V}} \text{ [R.A.CTRL.TO.ENV]}$$

$$\frac{b \notin locs(Q) \qquad b \neq a}{a(P)[Q \mid b.r\widetilde{V}] \rightarrow a(P \mid \mathsf{o}(b, r, \widetilde{V}))[Q]} \text{ [R.A.OUT]}$$

Figure 6: Routing Rules (Addressed Messages)

$locs(\nu n.P) = locs(P) \setminus \{n\}$	$locs(PQ) = \emptyset$
$locs(\mathbf{a}(P)[Q]) = \mathbf{a}, locs(P), locs(Q)$	$locs(\langle J \triangleright Q \rangle) = \emptyset$
$locs(P \mid Q) = locs(P), locs(Q)$	$locs(([\mu = V]P, Q)) = \emptyset$
$locs(\mathsf{pass}\ V) = \emptyset$	$locs(V) = \emptyset$
$locs(\mathbf{0}) = \emptyset$	$locs(P_1, \ldots, P_q) = \emptyset$

Figure 7: Multiset of Active Localities

can be thus chosen by the programmer. We see that an addressed resource $a.r$ is composed of an address (a locality name), which may correspond to an IP address, and a resource name, which may correspond to a port number. This construction is similar to the high-level $c@a$ construct of Nomadic Pict [24]. The second step of remote communication is the actual sending of the message to the remote locality. This communication step might be direct independently of the relative positions of the message and the destination, as in the Join calculus, or it might involve several local steps, following the structure of localities to reach the destination, as in the Ambient calculus. We remark that these two models may coincide when considering a flat model of localities.

Just as in the Distributed Join calculus and in Ambient calculi, we retain the idea of hierarchically organized localities, a crucial feature for capturing the spatial and logical partitioning of control in distributed systems. We assume there is an infinite countable set of locality names, and we let a, b range over this set. New locality names are introduced and bound by the restriction operator $\nu a.P$. To define the routing rules, we write $locs(P)$ for the multiset of unrestricted *active localities* of P. This multiset is formally defined in figure 7. In the rest of this paper, we say that a process P is *active* in Q if Q is structurally equivalent to a process of the form $\mathbf{E}(P)$ for some evaluation context $\mathbf{E}$. Localities in the M-calculus provide the means to enforce some control on incoming and outgoing messages. This control may be arbitrarily complex, may require maintaining some state, and should be kept separate from the program running in the locality. For this reason, localities take the form $a(P)[Q]$ where a is the name of the locality, P is a process controlling the locality and its interactions with the environment, and Q is the content of the locality. The first role of the controller is to filter incoming and outgoing messages. To this end, we introduce two special resource names i and

$$\frac{r \notin dln(P) \qquad r \in dln(Q)}{a(P \mid r\widetilde{V})[Q] \to a(P)[Q \mid r\widetilde{V}]} \text{ [R.L.CTRL.TO.CONT]}$$

$$\frac{r \in dln(P) \qquad r \notin dln(Q)}{a(P)[Q \mid r\widetilde{V}] \to a(P \mid r\widetilde{V})[Q]} \text{ [R.L.CONT.TO.CTRL]}$$

Figure 8: Routing Rules (Local Messages)

$$\begin{array}{ll}
dln(PQ) = \emptyset & dln(\nu n.P) = dln(P) \setminus \{n\} \\
dln(\mathbf{a}(P)[Q]) = \emptyset & dln(\langle \mathbf{r}_1\widetilde{x_1} \mid \ldots \mid \mathbf{r}_q\widetilde{x_q} \triangleright P\rangle) = \{\mathbf{r}_1, \ldots, \mathbf{r}_q\} \\
dln(\mathtt{pass}\ V) = \emptyset & dln(P \mid Q) = dln(P) \cup dln(Q) \\
dln(V) = \emptyset & dln(([\mu = V]P, Q)) = \emptyset \\
dln(\mathbf{0}) = \emptyset & dln((P_1, \ldots, P_q)) = \emptyset
\end{array}$$

Figure 9: Defined local names

o on which incoming and outgoing messages are intercepted (rules R.A.IN and R.A.OUT in figure 6). On interception, an incoming or outgoing message is split into three parts: the destination address, the targeted resource, and the message arguments. The controller should provide a reaction rule for these filtering channels, implementing the desired behavior. For instance, a locality that does not want to block any message could contain the process *Fwd* in its controller, where:

$$Fwd \stackrel{\text{def}}{=} \langle \mathsf{i}(x,y,z) \triangleright x.y\ z\rangle \mid \langle \mathsf{o}(x,y,z) \triangleright x.y\ z\rangle$$

We remark that this definition is stateless and relies on the other routing rules to send the message to its final destination automatically: even though the routing is step by step, it is not necessary to specify how to take each step. This is much different from the Ambient calculus where an explicit path to the target ambient needs to be given.

As localities form a tree, there is no notion of site as in Nomadic Pict (a site can be modeled by a locality at a given level in the tree) and the routing algorithm is part of our semantics. It is however possible to express different routing algorithms by forwarding messages from locality to locality using some specified resource. One example of this is the simulation of the dynamic Join calculus in the M-calculus in [19]. It is also possible to intercept and reroute messages using the control mechanism, as shown below.

A message present in the controller is considered as having been controlled and may freely leave the controller (rules R.A.CTRL.TO.CONT and R.A.CTRL.TO.ENV). A message that has reached its final destination becomes a local message (rules R.A.CTRL.FINAL and R.A.CONT.FINAL). Local messages may move freely from controller to content and vice versa (figure 8), depending on where the resource is defined. To this end, we call *defined local names* the set of resources that are defined in a given process without inspecting sublocalities. This set is formally defined in figure 9. We give an example of transparent incoming message routing in figure 10.

One interesting feature when writing a reaction rule for the filtering channels i and o is to be able to test the target locality or resource. To this end, we introduce a simple name matching operator $([\mu = V]P, Q)$, whose semantics (rules R.IF.THEN and R.IF.ELSE in figure 4) rely on a *match*() predicate, which is true only in the following cases:

$$match(\_, V) \qquad\qquad match(n, n)$$

$$\begin{array}{ll}
& \underline{a.r\widetilde{V}} \mid b(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)\,[\,\underline{a}(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)[\langle r = \ldots\rangle]\,] \\
\to & b(\underline{\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle} \mid \underline{\mathsf{i}(a,r,\widetilde{V})})\,[\,a(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)[\langle r = \ldots\rangle]\,] \\
\to & b(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle \mid \underline{a.r\widetilde{V}})\,[\,\underline{a}(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)[\langle r = \ldots\rangle]\,] \\
\to & b(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)\,[\,\underline{a.r\widetilde{V}} \mid \underline{a}(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)[\langle r = \ldots\rangle]\,] \\
\to & b(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)\,[\,a(\underline{\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle} \mid \underline{\mathsf{i}(a,r,\widetilde{V})})[\langle r = \ldots\rangle]\,] \\
\to & b(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)\,[\,\underline{a}(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle \mid \underline{a.r\widetilde{V}})[\langle r = \ldots\rangle]\,] \\
\to & b(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)\,[\,a(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle \mid \underline{r\widetilde{V}})[\underline{\langle r = \ldots\rangle}]\,] \\
\to & b(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)\,[\,a(\langle \mathsf{i}(d,r,v) \triangleright d.rv\rangle)[\langle r = \ldots\rangle \mid r\widetilde{V}]\,]
\end{array}$$

Figure 10: Remote communication example

For instance, a filter for incoming messages could be:

$$\langle \mathsf{i}(d,r,v) \triangleright [a = d](b.r\ v, \mathbf{0})\rangle$$

This filter throws away any message that is not for locality a, and reroutes messages for a to b.

As in the Ambient calculus and the Join calculus, we provide a way to modify the tree structure of localities. However, we want to be able to control incoming and outgoing localities at every locality boundary, as is possible with remote communication. We thus unify migration and remote communication by considering migration as the communication of a frozen process. A frozen process is of the form $\lambda.P$, and may be unfrozen by applying it to the null value (). We actually consider a generalization by embedding a call by value λ-calculus within our calculus (with the usual β reduction rule R.BETA of figure 4). We use standard notational conventions: in a term $\lambda x.P$ or $\nu n.P$, the scope extends as far to the right as possible; $PQ_1 \ldots Q_n$ stands for $(\ldots(PQ_1)\ldots Q_n)$; $\lambda x_1 \ldots x_q.P$ stands for $\lambda x_1.\ldots\lambda x_q.P$; $\nu n_1 \ldots n_q.P$ stands for $\nu n_1.\ldots\nu n_q.P$. We also make use of the notation $\lambda.P$ to stand for a *thunk* $\lambda x.P$, with x not free in P.

The *passivation* primitive $\mathtt{pass}\ V$, where V is a function expecting a locality name and two frozen processes, is introduced to freeze running processes. Passivation is the second role of a controller: when evaluated in the controller of a locality $a(\mathtt{pass}\ V \mid P)[Q]$, the locality is split into three parts: its name a, its frozen controller $\lambda.P$, and its frozen content $\lambda.Q$. These parts are given as arguments to the function V:

$$a(\mathtt{pass}\ V \mid P)[Q] \to V\ a\ (\lambda.P)\ (\lambda.Q)$$

For instance, a function $V = \lambda xpq.x(p())[q()]$ simply recreates the passivated locality.

Locality mobility can be implemented using higher-order messages and passivation (cf the go construct in the Join calculus and ambients `in` and `out` capabilities). Locality $Q^m(a)$ below can be moved to a different locality:

$$\begin{array}{rcl}
Q^m(a) & = & a(Fwd \mid \langle \mathsf{go}\, u \triangleright Go(u)\rangle)[Q] \\
Go(u) & = & \mathtt{pass}\ \lambda x\, p\, q.(u.\mathsf{enter}\ \lambda.x(p())[q()])
\end{array}$$

A request $\mathsf{go}\, b$, results in the passivation of the locality a and its sending as a thunk to the resource enter of the locality named b. If the request comes from the outside of the locality, the result is an objective form of move. If the request comes from the content of the locality, the result is a subjective form of move. The controller of locality b can contain the process *Enter*, depicted below, to define the resource enter used to insert and spawn a frozen process in its content:

$$Enter \quad = \quad \langle \mathsf{enter}\, f \triangleright \mathtt{pass}\ \lambda x\, p\, q.x(p())[q() \mid f()]\rangle$$

Passivation may also be used to implement various forms of control on a locality. Locality $Q^o(a)$ below can be suspended, resumed, dissolved (cf the open capability of ambients), and updated with a new controller (we note simply r a message of the form $r()$):

$$\begin{array}{rcl} Q^o(a) & = & \nu s\, on.L(a,s,on) \\ L(a,s,on) & = & a \left(\begin{array}{l} Fwd \mid \langle \mathsf{suspend} \mid on \triangleright S(s)\rangle \\ \mid \langle \mathsf{resume} \mid s\, f \triangleright R(f,on)\rangle \\ \mid \langle \mathsf{open} \triangleright O\rangle \mid \langle \mathsf{update}\, f \triangleright U(f)\rangle \mid on \end{array} \right) [Q] \\ S(s) & = & \mathtt{pass}\ \lambda x\, p\, q.x(p() \mid (s\, q))[\mathbf{0}] \\ R(f,on) & = & \mathtt{pass}\ \lambda x\, p\, q.x(p() \mid on)[f()] \\ O & = & \mathtt{pass}\ \lambda x\, p\, q.q() \\ U(f) & = & \mathtt{pass}\ \lambda x\, p\, q.x(f())[q()] \end{array}$$

As usual, we take the reduction relation for the M-calculus, $\rightarrow$, as the smallest relation that satisfies the rules given in Figures 4, 6, and 8.

We present as an example an M-calculus variant of the taxi example presented by L. Cardelli in [3]. The basic idea is as follows: a passenger enters a taxi by specifying a route (list of localities to enter), a sitting behavior (to be executed while sitting in the taxi), and a continuation behavior (to be executed on arrival) – L. Cardelli's version only comprises a route and a continuation behavior.

A route consists of a list of locality names $a_1, \ldots, a_n$, which we represent by the frozen process

$$M \quad = \quad \begin{array}{l} \lambda.\mathtt{route}(a_1, \\ \quad \lambda.\mathtt{route}(a_2, \\ \qquad \ldots, \\ \qquad\quad \lambda.\mathtt{route}(a_n, \lambda.\mathtt{route}(\mathtt{nil}, \lambda.\mathbf{0})) \\ \ldots)) \end{array}$$

where nil is a special locality name reserved to signal the end of the list.

A passenger requests entrance in the taxi named tx by emitting a message

$$\mathtt{tx.enter}(M, \lambda.S, \lambda.C)$$

where S is the sitting behavior, and C is the continuation behavior.

A taxi named tx is defined as follows:

$$\begin{array}{rcl} \mathtt{Taxi} & = & \nu \mathtt{f}.\mathtt{tx}(\langle \mathtt{Enter}\rangle \mid \langle \mathtt{Exit}\rangle \mid \langle \mathtt{Route}\rangle \mid \mathtt{f})[\mathbf{0}] \\ \mathtt{Enter} & = & \begin{array}{l} \mathtt{enter}(r,s,c) \mid \mathtt{f} \triangleright \\ \quad \mathtt{pass}\ \lambda x\, p\, q.x(p() | \mathtt{onexit}(c) | r())[s()] \end{array} \\ \mathtt{Exit} & = & \begin{array}{l} \mathtt{exit} \mid \mathtt{onexit}(c) \triangleright \\ \quad \mathtt{pass}\ \lambda x\, p\, q.x(p() | \mathtt{f})[\mathbf{0}] \mid q() \mid c() \end{array} \\ \mathtt{Route} & = & \begin{array}{l} \mathtt{route}(x,y) \triangleright \\ \left(\begin{array}{l} [\mathtt{nil} = x] \\ \quad \mathtt{exit}, \\ \quad \mathtt{pass}\ \lambda z\, p\, q. \\ \qquad x.\mathtt{visit}(\lambda.z(y() \mid p())[q()]) \end{array} \right) \end{array} \end{array}$$

Intuitively, each taxi contains a private lock (here the name f) indicating the taxi is free to hire. When a passenger enters the taxi, the lock is consumed, the frozen continuation is stored in the reference cell onexit, the route is unfrozen, and the sitting behaviour is spawned in the content of the taxi, using passivation. Since the route consists of a message containing a locality name and the rest of the route, the ⟨Route⟩ definition matches the locality name. If if it is nil, then the taxi has arrived, and the exit message is spawned. This message triggers the ⟨Exit⟩ definition, that consumes it as well as the reference cell onexit, recreates the taxi as free to hire (spawning the lock f), and spawns the continuation as well as the content of the taxi outside. If the next destination is not nil, then the taxi migrates to this destination. We suppose every locality has a channel visit that simply spawns its argument. The frozen taxi is thus sent to the next stop, with the remaining of the route spawned in its controller.

τ	$::=$		type
		Δ	process type
	$\mid$	σ	value type
σ	$::=$		value type
		$\mathtt{unit}$	unit type
	$\mid$	α	plain type variable
	$\mid$	$\mathtt{dom}(w)$	name type
	$\mid$	$\sigma_1, \ldots, \sigma_q$	tuple type
	$\mid$	$\sigma \rightarrow \tau$	function type
	$\mid$	$\langle\sigma\rangle_\Delta$	plain resource
	$\mid$	$\langle\sigma\rangle_\Delta^+$	sendable resource
w	$::=$		locality name variable
		a	locality name
	$\mid$	δ	name type variable
	$\mid$	$\emptyset$	no such locality
Δ	$::=$		locality name multiset
		$\emptyset$	empty multiset
	$\mid$	ρ	multiset variable
	$\mid$	δ	name type variable
	$\mid$	a	locality name
	$\mid$	Δ, Δ	multiset union
s	$::=$	$\forall\widetilde{\alpha}\widetilde{\delta}\widetilde{\rho}.\sigma$	type scheme

Figure 11: Types: Syntax

3. THE TYPE SYSTEM

The routing rules for addressed messages in Figure 6 rely heavily on locality names and active localities. Locality names thus play the role of *addresses* in the M-calculus. However, to faithfully mirror the situation in current wide area networks, and to allow for an effective implementation of the calculus, one must ensure the determinacy of the final destination of a remote message by ensuring that no two active localities may bear the same name. Unfortunately, it is not possible to obtain this property by syntactic means only. In fact, even a simple type system will not do because of the higher-order features of the calculus. In presence of the passivation operator, one must indeed be careful of the effects of functions on locality names. For instance, if a resource twice is defined as $\langle \mathsf{twice}\, f \triangleright f() \mid f()\rangle$, then a passivation instruction of the form $\mathtt{pass}\ \lambda x\, p\, q.(\mathsf{twice}\, \lambda.x(p())[q()])$ may lead to the illicit duplication of the passivated cell.

To enforce the unicity of active locality names, we introduce the following type system. The grammar for types is given in Figure 11. We remark that terms in the M-

$$
\begin{array}{lrcl}
 & \Delta \leq \Delta' & \Leftarrow & \Delta \subseteq \Delta' \\
 & \widetilde{\sigma} \leq \widetilde{\sigma'} & \Leftarrow & (\sigma_i \leq \sigma_i')^{i \in [1..n]} \\
\mathtt{unit} \leq \mathtt{unit} & \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau' & \Leftarrow & \sigma' \leq \sigma \text{ and } \tau \leq \tau' \\
\mathtt{dom}(w) \leq \mathtt{dom}(w) & \langle\sigma\rangle_\Delta \leq \langle\sigma'\rangle_{\Delta'} & \Leftarrow & \sigma' \leq \sigma \text{ and } \Delta \leq \Delta' \\
\alpha \leq \alpha & \langle\sigma\rangle_\Delta \leq \sigma' \rightarrow \Delta' & \Leftarrow & \sigma' \leq \sigma \text{ and } \Delta \leq \Delta' \\
\langle\sigma\rangle_\Delta^+ \leq \langle\sigma\rangle_\Delta^+ & \langle\sigma\rangle_\Delta^+ \leq \langle\sigma'\rangle_{\Delta'} & \Leftarrow & \sigma' \leq \sigma \text{ and } \Delta \leq \Delta' \\
 & \langle\sigma\rangle_\Delta^+ \leq \sigma' \rightarrow \Delta' & \Leftarrow & \sigma' \leq \sigma \text{ and } \Delta \leq \Delta'
\end{array}
$$

Figure 12: Subtyping relation

calculus are partitioned in two kinds: *processes* and *expressions*. This distinction is difficult to manifest by relying only on the syntax, mainly because of functional application (the result of an application may either be an expression or a process), but the intuition behind this partition is that processes may be put in parallel, and include messages, localities, controller and content of localities, whereas expressions may reduce to values and include functions, tuples, and names. We formalize this partition by making a distinction among types τ between *process types* Δ and *value types* σ. Process types are multisets of locality names, representing an upper bound of the localities that may be or become active in the process. Value types represent the value the expression may eventually reduce to. They include function types $\sigma \rightarrow \tau$, tuple types $\widetilde{\sigma}$, the $\mathtt{unit}$ type, and types for names. Resource names have type $\langle\sigma\rangle_\Delta$ or $\langle\sigma\rangle_\Delta^+$ if they expect an argument of type σ (which may be a tuple) and if a message on this resource name leads to the creation of localities Δ. A resource name with the type $\langle\sigma\rangle_\Delta^+$ may be received and used for further input, as in the creation of a reaction rule $\langle \mathit{create}\ x \triangleright \langle x() \triangleright a(\mathbf{0})[\mathbf{0}]\rangle\rangle$, where *create* may have the type $\langle\langle\mathtt{unit}\rangle_a^+\rangle_\emptyset$. Locality names have type $\mathtt{dom}(w)$, where w may either be a locality name a, a name type variable δ, or the empty set $\emptyset$. Intuitively, a locality name a has singleton type $\mathtt{dom}(a)$, a variable x may have type $\mathtt{dom}(\delta)$, and no name may have type $\mathtt{dom}(\emptyset)$ (this last type is needed for technical reasons). Name type variables δ reflect term variables x that may be instantiated to locality names, as in $\langle \mathit{create}(x) \triangleright x(\mathbf{0})[\mathbf{0}]\rangle$ where *create* may be given the type $\forall\delta.\langle\mathtt{dom}(\delta)\rangle_\delta$.

Multisets Δ include locality names a, name type variables δ, multiset variables ρ, the empty multiset $\emptyset$, and unions of multisets Δ, Δ'. The basic intuition to guarantee the unicity of names of active localities in a process P is to type a process P with a multiset Δ. If this multiset happens to be a set, then we prove that every active locality bears a unique name.

We use $\forall\widetilde{\alpha}\widetilde{\delta}\widetilde{\rho}.\sigma$ to denote a type scheme where plain type variables $\widetilde{\alpha}$, name type variable $\widetilde{\delta}$ and multiset variables $\widetilde{\rho}$ are generalized. We use β to range indifferently over these different type variables.

In what follows, we consider an extended syntax for the M-calculus, where new resource names are required to be annotated by their type scheme, in order to specify whether the resource has a plain resource type (which may be polymorphic) or if it has a sendable resource type (which cannot be polymorphic for safety reasons).

The intersection operation between multisets, $\cap$, is the standard intersection on multisets (taking the smallest number of occurrences in both multisets). The inclusion relation $\subseteq$ between multisets is also the standard one. By $\Delta - \Delta'$, we denote the multiset which is composed of the elements of Δ (locality names or multiset variables) after removing each element of Δ'. For instance $\rho, \rho, a, b, b - \rho, a, a, b = \rho, b$.

$$
\begin{array}{rcl}
a, \Delta \sqcup a, \Delta' &=& a, (\Delta \sqcup \Delta') \\
\rho, \Delta \sqcup \rho, \Delta' &=& \rho, (\Delta \sqcup \Delta') \\
\delta, \Delta \sqcup \delta, \Delta' &=& \delta, (\Delta \sqcup \Delta') \\
\Delta \sqcup \Delta' &=& \Delta, \Delta' \text{ if } \Delta \cap \Delta' = \emptyset \\
\mathtt{unit} \sqcup \mathtt{unit} &=& \mathtt{unit} \\
\mathtt{dom}(w) \sqcup \mathtt{dom}(w) &=& \mathtt{dom}(w) \\
\alpha \sqcup \alpha &=& \alpha \\
(\sigma_1, \ldots, \sigma_n) \sqcup (\sigma_1', \ldots, \sigma_n') &=& (\sigma_1 \sqcup \sigma_1', \ldots, \sigma_n \sqcup \sigma_n') \\
\sigma \rightarrow \tau \sqcup \sigma' \rightarrow \tau' &=& (\sigma \sqcap \sigma') \rightarrow (\tau \sqcup \tau') \\
\langle\sigma\rangle_\Delta \sqcup \langle\sigma'\rangle_{\Delta'} &=& \langle\sigma \sqcap \sigma'\rangle_{\Delta \sqcup \Delta'} \\
\langle\sigma\rangle_\Delta \sqcup \sigma' \rightarrow \Delta' &=& (\sigma \sqcap \sigma') \rightarrow (\Delta \sqcup \Delta') \\
\langle\sigma\rangle_\Delta^+ \sqcup \langle\sigma\rangle_\Delta^+ &=& \langle\sigma\rangle_\Delta^+ \\
\langle\sigma\rangle_\Delta^+ \sqcup \langle\sigma'\rangle_{\Delta'}^+ &=& \langle\sigma \sqcap \sigma'\rangle_{\Delta \sqcup \Delta'} \text{ if } \sigma \neq \sigma' \text{ or } \Delta \neq \Delta' \\
\langle\sigma\rangle_\Delta^+ \sqcup \langle\sigma'\rangle_{\Delta'} &=& \langle\sigma \sqcap \sigma'\rangle_{\Delta \sqcup \Delta'} \\
\langle\sigma\rangle_\Delta^+ \sqcup \sigma' \rightarrow \Delta' &=& (\sigma \sqcap \sigma') \rightarrow (\Delta \sqcup \Delta')
\end{array}
$$

Figure 13: Definition of $\sqcup$

$$
\begin{array}{rcl}
a, \Delta \sqcap a, \Delta' &=& a, (\Delta \sqcap \Delta') \\
\rho, \Delta \sqcap \rho, \Delta' &=& \rho, (\Delta \sqcap \Delta') \\
\delta, \Delta \sqcap \delta, \Delta' &=& \delta, (\Delta \sqcap \Delta') \\
\Delta \sqcap \Delta' &=& \emptyset \text{ if } \Delta \cap \Delta' = \emptyset \\
\mathtt{unit} \sqcap \mathtt{unit} &=& \mathtt{unit} \\
\mathtt{dom}(w) \sqcap \mathtt{dom}(w) &=& \mathtt{dom}(w) \\
\alpha \sqcap \alpha &=& \alpha \\
(\sigma_1, \ldots, \sigma_n) \sqcap (\sigma_1', \ldots, \sigma_n') &=& (\sigma_1 \sqcap \sigma_1', \ldots, \sigma_n \sqcap \sigma_n') \\
\sigma \rightarrow \tau \sqcap \sigma' \rightarrow \tau' &=& (\sigma \sqcup \sigma') \rightarrow (\tau \sqcap \tau') \\
\langle\sigma\rangle_\Delta \sqcap \langle\sigma'\rangle_{\Delta'} &=& \langle\sigma \sqcup \sigma'\rangle_{\Delta \sqcap \Delta'} \\
\langle\sigma\rangle_\Delta \sqcap \sigma' \rightarrow \Delta' &=& \langle\sigma \sqcup \sigma'\rangle_{\Delta \sqcap \Delta'} \\
\langle\sigma\rangle_\Delta^+ \sqcap \sigma' &=& \langle\sigma\rangle_\Delta^+ \text{ if } \langle\sigma\rangle_\Delta^+ \leq \sigma'
\end{array}
$$

Figure 14: Definition of $\sqcap$

We define in figure 12 a subtyping relation $\leq$ (where $\widetilde{\sigma}$ and $\widetilde{\sigma'}$ are tuples of the same size n).

The intuition behind the subtyping relation is that it is safe (with regard to the unicity of locality names) to replace a process that includes more active localities with a process that includes fewer active localities. It is also safe to replace a function by a resource name if the types agree (a resource name may be used to send a message, by functional application, but it can also be used to create an addressed resource name). It is also safe to replace a plain resource name by a sendable resource name (sendable resource names may be used for message sending, but they can also be used to instantiate variables that are defined names of a Join pattern).

We remark that sendable resource types have no subtype, except themselves. This is necessary for type safety since, unlike [25], we do not distinguish between input and output types.

We define the symmetric $\sqcup$ and $\sqcap$ operators on types in figures 13 and 14 (other possible cases not listed are undefined). $\Delta_1 \sqcup \Delta_2$ is defined as the multiset where the multiplicity of each name a is taken to be the *max* of the multiplicities in Δ_1 and Δ_2, and $\Delta_1 \sqcap \Delta_2$ is defined as the multiset where the multiplicity of each name a is taken to be the *min* of the multiplicities in Δ_1 and Δ_2 (it is the usual intersection on multisets).

We use Γ and its decorated variants to denote type environments, i.e. finite mappings between names and type schemes. We define the set of free plain type variables *ftv*()

$$
\begin{array}{rlrl}
fsv(\emptyset) = \emptyset & & fwv(\emptyset) = \emptyset \\
fsv(\rho) = \{\rho\} & & fwv(\rho) = \emptyset \\
fsv(\delta) = \emptyset & & fwv(\delta) = \{\delta\} \\
fsv(a) = \emptyset & & fwv(a) = \emptyset \\
fsv(\Delta, \Delta') = fsv(\Delta) \cup fsv(\Delta') & & fwv(\Delta, \Delta') = fwv(\Delta) \cup fwv(\Delta')
\end{array}
$$

Figure 15: Definition of *fsv*() and *fwv*()

$$
\begin{array}{rcl}
C & ::= & \cdot : \tau \mid \nu r : s.C \mid \nu a.C \mid \lambda x.C \mid \mathbf{a}(C)[Q] \\
& \mid & \mathbf{a}(P)[C] \mid (C \mid P) \mid (P \mid C) \mid \mathtt{pass}\ C \mid (CQ) \\
& \mid & (PC) \mid ([\mu = C]P, Q) \mid ([\mu = V]C, P) \\
& \mid & ([\mu = V]P, C) \mid \langle J \triangleright C \rangle \mid (P_1, \ldots, C, \ldots, P_q)
\end{array}
$$

Figure 16: Typed Contexts

as usual. We define the set of free multiset variables *fsv*() and the set of free name type variables *fwv*() in figure 15.

We let the set of free variables *fv*() be the union of *ftv*(), *fsv*(), and *fwv*(). Type judgments take the following form, where C is an M-context extended with a typed hole, as defined in Figure 16: $\Gamma \vdash C : \tau$ and $\Gamma \vdash P : \tau$.

We write $fun(\sigma; \tau)$ for the types of all values that may be used as functions, that is for either $\sigma \rightarrow \tau$, $\langle\sigma\rangle_\tau$, or $\langle\sigma\rangle^+_\tau$ (in the latter two cases, τ is necessarily a process type Δ). We write $chan(\sigma; \Delta)$ for channel types (either $\langle\sigma\rangle_\Delta$ or $\langle\sigma\rangle^+_\Delta$).

In order to correctly type the i and o resources, we consider only type environments with the following associations:

$$
\begin{array}{rcl}
\mathsf{i} & : & \forall\alpha\delta\rho.\langle \mathtt{dom}(\delta), \langle\alpha\rangle_\rho, \alpha\rangle_\rho \\
\mathsf{o} & : & \forall\alpha\delta\rho.\langle \mathtt{dom}(\delta), \langle\alpha\rangle_\rho, \alpha\rangle_\rho
\end{array}
$$

Both the i and o resources expect a locality name (the destination of the message), the targeted resource expecting an argument of type α and creating localities ρ, and an argument of type α. A message on such a channel potentially creates localities ρ.

The type system is defined by the rules in Figure 17. They make use of the *Inst* operator, that takes a type scheme and returns a type where the generalized plain type variables, multiset variables, and name type variables have been instantiated to types, multisets, and locality names or name type variables respectively.

Typing rule Join may seem complex but is the usual typing rule for Join patterns: the guarded process is typed in an environment extended with the formal parameters, and the result is checked to create fewer localities than advertised by the resource types. Every defined resource name that is a variable is checked to have a sendable resource type in the environment. The additional hypotheses check that the type schemes associated with the resources are consistent with the typing environment, following the usual rules of the Distributed Join calculus: no generalized variable may occur free in the environment nor be shared by two resources in a Join pattern. In rule Pass, the passivation function is checked to have a type that is a subtype of a function expecting a locality name and two thunks. The name type variable δ in the locality name type represents the name of the passivated cell. Each thunk type includes a multiset variable ρ_1 or ρ_2 representing the active localities of the thunk. These three variables are intuitively generalized by checking that they do no occur in the typing environment (they will respectively be substituted by the name of the passivated locality and by the active localities of the controller and content, that are unknown when typing the primitive). The type of the whole passivation construct is simply the type of the result of the passivation function, removing the name of the passivated locality and the multiset variables since the locality is passivated.

The soundness of the our type system is characterized by the following definitions and theorems.

Definition 1. A typing environment Γ is *well-formed* if and only if:

1. Γ only contains associations of the form $r : \forall\widetilde{\beta}.\langle\sigma\rangle_\Delta$, $r : \langle\sigma\rangle^+_\Delta$, and $a : \mathtt{dom}(a)$;
2. we have $fn(\Gamma) = dom(\Gamma)$.

Lemma 3.1. *Let $\Gamma \vdash P : \tau$ be a typing derivation with Γ well-formed. If $P \equiv P'$, then there exists a typing derivation $\Gamma \vdash P' : \tau$.*

Theorem 1 (Subject reduction). *Let $\Gamma \vdash P : \tau$ be a typing derivation with Γ well-formed. If $P \rightarrow P'$, then there exists a type τ' such that $\tau' \leq \tau$ and $\Gamma \vdash P' : \tau'$.*

Definition 2. A locality $a(P')[Q']$ is said to be *free and active* in P if it is active in P and if it is not under a scope restriction for a.

A process P has *failed* if and only if there is some process P' active in P containing at least two free and active localities bearing the same name.

Theorem 2 (Progress). *Let $\Gamma \vdash P : \Delta$ be a typing derivation with Γ well-formed. If Δ is a set, then the process P has not failed.*

Most of the complexity of the subject reduction proof is alleviated by distinguishing name variables x from name type variables δ, greatly simplifying the substitution lemma. Complete proofs are available in the draft of the long version of this paper [18].

Our notion of progress deals only with the unicity of active locality names. Since our type system is very close to type systems for the λ-calculus and the Distributed Join calculus, this progress property can be easily extended to more usual guarantees.

We now discuss the power and limits of our type system.

Linearity. Since our type system aims at enforcing the linearity of names of active localities, we describe how this feature interacts with functions (or resources) that are not linear. First of all, our system guarantees the linearity of names of localities that are *active* (as described in theorem 2). Relaxing the linearity constraint for localities that are not active yields a much simpler type system (e.g. our system allows the typing of $\lambda.a(\mathbf{0})[\mathbf{0}] \mid a(\mathbf{0})[\mathbf{0}]$ as long as this function is not applied). We also require linearity of a newly introduced locality name in the process under the scope restriction (rule Nu.Dom, since the introduced name should not occur in the type). We recall that the type of a process is a conservative approximation of the active localities it may contain. In the following, we only consider processes that are active.

$$\frac{u : \forall\widetilde{\beta}.\sigma \in \Gamma \qquad \sigma\theta = \mathit{Inst}(\forall\widetilde{\beta}.\sigma) \qquad \mathit{fn}(\mathit{ran}(\theta)) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash u : \sigma\theta}\ [\textsc{Name}] \qquad \frac{}{\Gamma \vdash \mathbf{0} : \emptyset}\ [\textsc{Nil}] \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}}\ [\textsc{Void}]$$

$$\frac{}{\Gamma \vdash (\cdot : \tau) : \tau}\ [\textsc{Proc.Hole}] \qquad \frac{\Gamma \vdash \mathbf{a} : \mathtt{dom}(w) \qquad \Gamma \vdash \mathbf{r} : \mathit{chan}(\sigma; \Delta)}{\Gamma \vdash \mathbf{a}.\mathbf{r} : \sigma \to \Delta}\ [\textsc{Addr}]$$

$$\frac{\Gamma + x : \sigma \vdash P : \tau \qquad x \notin \mathit{dom}(\Gamma) \qquad \mathit{fn}(\sigma) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash \lambda x.P : \sigma \to \tau}\ [\textsc{Fun}] \qquad \frac{(\Gamma \vdash P_i : \sigma_i)^{i \in [1..q]}}{\Gamma \vdash P_1, \ldots, P_q : (\sigma_1, \ldots, \sigma_q)}\ [\textsc{Tuple}]$$

$$\frac{\Gamma \vdash \mathbf{a} : \mathtt{dom}(w) \qquad \Gamma \vdash P : \Delta_1 \qquad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash \mathbf{a}(P)[Q] : w, \Delta_1, \Delta_2}\ [\textsc{Dom}] \qquad \frac{\Gamma \vdash P : \Delta_1 \qquad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash P \mid Q : \Delta_1, \Delta_2}\ [\textsc{Par}]$$

$$\frac{\Gamma + r : \forall\widetilde{\beta}.\langle\sigma\rangle_\Delta \vdash P : \Delta_1 \qquad r \notin \mathit{dom}(\Gamma) \qquad \mathit{fv}(\forall\widetilde{\beta}.\langle\sigma\rangle_\Delta) = \emptyset \qquad \mathit{fn}(\forall\widetilde{\beta}.\langle\sigma\rangle_\Delta) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash \nu r : \forall\widetilde{\beta}.\langle\sigma\rangle_\Delta.P : \Delta_1}\ [\textsc{Nu.Res.1}]$$

$$\frac{\Gamma + r : \langle\sigma\rangle^+_\Delta \vdash P : \Delta_1 \qquad r \notin \mathit{dom}(\Gamma) \qquad \mathit{fv}(\langle\sigma\rangle^+_\Delta) = \emptyset \qquad \mathit{fn}(\langle\sigma\rangle^+_\Delta) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash \nu r : \langle\sigma\rangle^+_\Delta.P : \Delta_1}\ [\textsc{Nu.Res.2}]$$

$$\frac{\Gamma + a : \mathtt{dom}(a) \vdash P : \Delta \qquad a \notin \mathit{fn}(\Gamma) \qquad a \notin (\Delta - a)}{\Gamma \vdash \nu a.P : \Delta - a}\ [\textsc{Nu.Dom}]$$

$$\frac{\Gamma \vdash V : \mathtt{dom}(\delta) \to (\mathtt{unit} \to \Delta_1) \to \mathit{fun}(\mathtt{unit} \to \Delta_2; \Delta) \qquad \rho_1 \neq \rho_2 \qquad \rho_1 \in \Delta_1 \ \ \rho_2 \in \Delta_2 \qquad \delta, \rho_1, \rho_2 \notin \mathit{fv}(\Gamma) \cup (\Delta - (\delta, \rho_1, \rho_2))}{\Gamma \vdash \mathtt{pass}\ V : \Delta - (\delta, \rho_1, \rho_2)}\ [\textsc{Pass}]$$

$$\frac{\Gamma \vdash P : \mathit{fun}(\sigma; \tau) \qquad \sigma' \leq \sigma \qquad \Gamma \vdash Q : \sigma'}{\Gamma \vdash PQ : \tau}\ [\textsc{App}] \qquad \frac{\Gamma \vdash V : \tau \qquad \Gamma \vdash P : \tau_1 \qquad \Gamma \vdash Q : \tau_2}{\Gamma \vdash ([\mu = V]P, Q) : \tau_1 \sqcup \tau_2}\ [\textsc{Test}]$$

$$\frac{\begin{array}{c}(\mathbf{r}_i \text{ is not a variable and } r_i : s_i = \forall\widetilde{\beta_i}.\langle\widetilde{\sigma_i}\rangle_{\Delta_i} \in \Gamma, \text{ or } \mathbf{r}_i : s_i = \langle\widetilde{\sigma_i}\rangle^+_{\Delta_i} \in \Gamma)^{i \in [1..n]} \qquad \Delta' \leq \Delta_1, \ldots, \Delta_n \qquad (\widetilde{x_i})^i \cap \mathit{dom}(\Gamma) = \emptyset \\ \Gamma + \widetilde{x_1} : \widetilde{\sigma_1} + \ldots + \widetilde{x_n} : \widetilde{\sigma_n} \vdash P : \Delta' \qquad \forall i \in [1..n].\widetilde{\beta_i} \cap \mathit{fv}(\Gamma) = \emptyset \qquad \forall i, j \in [1..n]^2.i \neq j \implies \widetilde{\beta_i} \cap \widetilde{\beta_j} = \emptyset\end{array}}{\Gamma \vdash \langle \mathbf{r}_1\widetilde{x_1} \mid \ldots \mid \mathbf{r}_n\widetilde{x_n} \triangleright P\rangle : \emptyset}\ [\textsc{Join}]$$

Figure 17: Typing rules

Consider for instance the function $\lambda x.x() \mid x()$. This function takes a frozen process and runs it twice. It is perfectly reasonable to apply it to a frozen process that does not release localities with free names, as in: $(\lambda x.x() \mid x())\lambda.\mathbf{0}$ or $(\lambda x.x() \mid x())\lambda.\nu a.a(\mathbf{0})[\mathbf{0}]$. In these cases, the function has type $(\mathtt{unit} \to \emptyset) \to \emptyset$. The type system would reject applying this function to the thunk $\lambda.a(\mathbf{0})[\mathbf{0}]$.

A function $\lambda x.x()$, that only runs a process, can be given the type $(\mathtt{unit} \to \Delta) \to \Delta$. Since functions never have polymorphic types, one can instead consider a resource run defined as $\langle \mathsf{run}(x) \triangleright x()\rangle$. Such a resource can be given the type $\forall\rho.\langle\mathtt{unit} \to \rho\rangle_\rho$: it simply recreates the active localities frozen in its argument. The check that these localities do not interfere with currently active localities is done when typing a message on run. We remark that our use of multiset variables is very similar to row variables described in [16].

Subtyping. Subtyping is explicitly used in typing rule App. Returning to the example of the run resource above, this resource can receive a frozen process as argument, as in $(\mathsf{run}\ \lambda.a.(\mathbf{0})[\mathbf{0}])$. It can also receive a resource name, as in $(\mathsf{run}\ \mathsf{create_a})$ with $\langle \mathsf{create_a}() \triangleright a(\mathbf{0})[\mathbf{0}]\rangle$, since $\langle\mathtt{unit}\rangle_a$ is a subtype of $\mathtt{unit} \to a$.

The subtyping used in typing rule Join gives some flexibility when defining a resource, since the type of the resource gives an upper bound on the localities that may be created. For instance, the special channel i has type $\forall\alpha\delta\rho.\langle\mathtt{dom}(\delta), \langle\alpha\rangle_\rho, \alpha\rangle_\rho$. Typical definitions for this channel are $\langle \mathsf{i}(d, r, \mathit{args}) \triangleright d.r\ \mathit{args}\rangle$, which simply sends the message, and $\langle \mathsf{i}(d, r, \mathit{args}) \triangleright \mathbf{0}\rangle$, which discards it. The typing of the second definition relies on the fact that this definition of i does not create all the localities it is allowed to.

Dependent Types and Polymorphism. Our type system does not allow dependent types, but simulates them using polymorphism and *name type variables* (type variables that represent locality names), since locality names may occur in types. For instance the resource new in the definition $\langle \mathsf{new}(x) \triangleright x(\mathbf{0})[\mathbf{0}]\rangle$ may be given the polymorphic type $\forall\delta.\langle\mathtt{dom}(\delta)\rangle_\delta$: it expects any locality name and creates a locality with this name. We remark that our solution is less powerful than dependent types, since it does not allow the typing of $(\lambda f.f\ \ a \mid f\ \ b)(\lambda x.x(\mathbf{0})[\mathbf{0}])$. However, polymorphism is powerful enough to let us type definitions of the form: $\langle \mathsf{create}(x, p, q) \triangleright x(p())[q()]\rangle$, where create is given the type $\forall\delta\rho_1\rho_2.\langle\mathtt{dom}(\delta), \mathtt{unit} \to \rho_1, \mathtt{unit} \to \rho_2\rangle_{\delta,\rho_1,\rho_2}$.

Passivation. One limitation of our type system comes from the passivation operator. Consider for instance the process: $a\,(\mathtt{pass}\ \lambda xpq.x(\mathbf{0})[b(\mathbf{0})[\mathbf{0}]])\,[b(\mathbf{0})[\mathbf{0}]]$. The $\mathtt{pass}$ operator freezes locality a and respawns it, discarding its controller and content, but adds a locality b in its content. Since the locality b that is added is first discarded, one would expect this process to be well typed. This is not the case because locality b is active, and may be sent somewhere else (see for instance the migration example in section 2). This is why the process $\mathtt{pass}\ \lambda xpq.x(\mathbf{0})[b(\mathbf{0})[\mathbf{0}]]$ has type b: this process creates an active locality b, independently of the controller and content that are passivated. On the other hand, the process $\texttt{pass}\ \lambda xpq.x(p())[q()]$ has type $\emptyset$ (it only recreates a locality that has been passivated) and the following process is well typed: $a\,(\texttt{pass}\ \lambda xpq.x(p())[q()])\,[b(\mathbf{0})[\mathbf{0}]]$.

Example. As a small example, we remark that the taxi example we describe at the end of section 2 is well typed with the following bindings:

$$
\begin{array}{rcl}
\texttt{enter} & : & \forall\rho_S,\rho_C.\langle \texttt{unit}\to\emptyset, \texttt{unit}\to\rho_S, \texttt{unit}\to\rho_C\rangle_{\rho_S,\rho_C} \\
\texttt{exit} & : & \langle\texttt{unit}\rangle_\emptyset \\
\texttt{onexit} & : & \forall\rho_C.\langle\texttt{unit}\to\rho_C\rangle_{\rho_C} \\
\texttt{route} & : & \forall\delta.\langle\texttt{dom}(\delta),\texttt{unit}\to\emptyset\rangle_\emptyset \\
\texttt{visit} & : & \forall\rho.\langle\texttt{unit}\to\rho\rangle_\rho
\end{array}
$$

4. SIMULATING THE JOIN CALCULUS

In this section, we show that we can simulate the Distributed Join calculus in a straightforward manner, thus retaining all its expressivity. However, we first check that the M-Calculus addresses the limitation of the Distributed Join calculus described in the introduction.

The lack of control over communication and mobility is directly addressed by the possibility to program *controllers* for each locality to filter incoming and outgoing messages. These controllers are normal M-Calculus processes, they can be arbitrarily complex.

Dynamic binding, or more precisely locality based binding, is provided by *addressed messages*, which explicitly include the name of the locality hosting the targeted resource, and by the possibility to define *resources* at several localities. A simulation of the Dynamic Join calculus [20] in the M-Calculus presented in [19] illustrates how to use addressed messages to get transparent locality based binding.

Finally, programmable controllers and the *passivation* operator let the programmer define localities with different semantics within the same program.

Since the M-calculus is a direct offspring of the Distributed Join calculus, a translation of a Distributed Join calculus process into an M-calculus process is relatively straightforward. Such a translation is interesting to present, however, because it illustrates the versatility of programmable localities. In the following, we only consider Join processes with no free names. We proceed in three steps, in order to account for the fact that resource names are introduced with their types in the M-calculus, and that message routing requires the messages to be annotated with address information.

The first step consists in typing the Distributed Join calculus process, and annotating every local Join calculus definition with the channel names (as well as their type schemes) and location names it defines. We write $\texttt{def}\ (D; n_1 : s_1, n_q : s_q, a_1, \ldots, a_r)\ \texttt{in} \ldots$ the result of the first translation step of $\texttt{def}\ D\ \texttt{in}\ P$ if $dn(D) = \{n_1, \ldots, n_q\} \cup \{a_1, \ldots, a_r\}$ and if the channel names n_i have type schemes s_i after generalization. By definition of the types of the Distributed Join calculus, the s_i are of the form $\forall\widetilde{\alpha}.\langle\widetilde{\tau}\rangle$. We suppose that these type schemes have no free type variables, and we transform them immediately in M-calculus type schemes, writing $\langle\rangle_\emptyset$ instead of $\langle\rangle$.

The second translation step consists in annotating every channel name that is not a variable with the locality where its definitions reside. To do this, we rely on the fact that in the Distributed Join calculus, the only locality in which a definition may eventually be dissolved is the syntactically enclosing locality. This property is a direct consequence of the definition of migration: the only way to have processes migrate in the Distributed Join calculus is by locality migration. Thus definitions cannot be separated from the locality where they syntactically occur. For reasons of space, we do not formally present the algorithm used in this second step, which is roughly of the following form: for every channel name that is not in a Join pattern, find the enclosing $\texttt{def}$ binder for this name, then find the name of the syntactically enclosing locality, and prepend the locality name to the channel name.

The third step is the translation to the M-calculus itself. We represent a location of the Distributed Join calculus $a[\,\cdot\,]$ by a locality $a(PJ)[\,\cdot\,]$ with:

$$
\begin{array}{rcl}
PJ & = & \mathit{Fwd} \mid \langle \mathsf{add}\ f \triangleright \mathit{Add}(f)\rangle \mid \langle \mathsf{go}\ (b,\kappa) \triangleright \mathit{Send}(b,\kappa)\rangle \\
\mathit{Add}(f) & = & \mathsf{pass}\ \lambda xpq.x(p())[q() \mid f()] \\
\mathit{Send}(b,\kappa) & = & \mathsf{pass}\ \lambda xpq.(b.\mathsf{add}\ \lambda.x(p())[q() \mid \kappa()])
\end{array}
$$

We type the resource add with the type $\forall\rho.\langle\texttt{unit}\to\rho\rangle_\rho$ that expects a thunk and frees it, and the resource go with the type $\forall\delta\rho.\langle\texttt{dom}(\delta),\texttt{unit}\to\rho\rangle_\rho$, that takes a locality name and a frozen continuation that it will eventually spawn.

Writing $[\![\,\cdot\,]\!]$ for the translation operator, we have:

$$
\begin{array}{rclrcl}
[\![b.n]\!] & = & b.n & [\![b]\!] & = & b \\
[\![x]\!] & = & x & [\![\mathbf{0}]\!] & = & \mathbf{0} \\
[\![P \mid Q]\!] & = & [\![P]\!] \mid [\![Q]\!] & [\![\top]\!] & = & \mathbf{0} \\
[\![\mathsf{go}\ b; P]\!] & = & \mathsf{go}(b, \lambda.[\![P]\!]) & [\![D, D']\!] & = & [\![D]\!] \mid [\![D']\!] \\
[\![b\,[D : P]\,]\!] & = & b(PJ)[[\![D]\!] \mid [\![P]\!]] & & &
\end{array}
$$
$$
\begin{array}{l}
[\![m\langle n_1, \ldots, n_q\rangle]\!] = [\![m]\!]([\![n_1]\!], \ldots, [\![n_q]\!]) \\
[\![n_1\langle\widetilde{x_1}\rangle \mid \ldots \mid n_q\langle\widetilde{x_q}\rangle \triangleright P]\!] = \langle n_1\langle\widetilde{x_1}\rangle \mid \ldots \mid n_q\langle\widetilde{x_q}\rangle \triangleright [\![P]\!]\rangle \\
[\![\texttt{def}\ (D; n_1 : s_1, \ldots, n_q : s_q, a_1, \ldots, a_r)\ \texttt{in}\ P]\!] = \\
\qquad \nu n_1 : s_1, \ldots, n_q : s_q.\nu a_1, \ldots, a_r.[\![D]\!] \mid [\![P]\!]
\end{array}
$$

Note that our translation is not entirely faithful with respect to migration, since in the Distributed Join calculus migration does not occur if the target locality is a sublocality of the moving one. In order to detect these cases, we would need a more complex translation.

We now extend the previous translation to the Distributed Join calculus with failures. To do this, we only need to change the translation of a locality, replacing the PJ controller by the following $PJF(a)$ controller:

$$
\begin{array}{rcl}
PJF & = & \quad \mathit{Fwd} \\
& & \mid \quad \langle \mathsf{add}\ f \triangleright \mathit{Add}(f)\rangle \\
& & \mid \quad \langle \mathsf{go}\ (b,\kappa) \triangleright \mathit{Send}(b,\kappa)\rangle \\
& & \mid \quad \langle \mathsf{halt} \triangleright \mathit{Halt}\rangle \\
& & \mid \quad \langle \mathsf{ping}\ (y,n) \triangleright y\,()\rangle \\
\mathit{Add}(f) & = & \mathsf{pass}\ \lambda xpq.x(p())[q() \mid f()] \\
\mathit{Send}(b,\kappa) & = & \mathsf{pass}\ \lambda xpq.(b.\mathsf{add}\ \lambda.x(p())[q() \mid \kappa()]) \\
\mathit{Halt} & = & \mathsf{pass}\ \lambda xpq.\ x \left(\begin{array}{ll} & \langle \mathsf{ping}\ (y,n) \triangleright n\,()\rangle \\ \mid & \langle \mathsf{add}\ f \triangleright \mathit{Add}(f)\rangle \\ \mid & \langle \mathsf{i}\ (d,m,v) \triangleright P(m,v)\rangle \\ \mid & \langle \mathsf{o}\ (d,m,v) = \mathbf{0}\rangle \end{array} \right) [q()] \\
P(m,v) & = & ([\mathsf{ping} = m]m\ v, ([\mathsf{add} = m]m\ v, \mathbf{0}))
\end{array}
$$

Thus, when a locality has failed, it prevents all outgoing messages from leaving the failed locality: no sublocality may send a message outside. Similarly, with respect to incoming messages, a failed locality only accepts messages sent on ping or on add for itself or its sublocalities. Messages on ping are emitted locally, and will be subsequently reduced by the new definition for `ping`, that will answer saying the sublocality has failed. Messages on `add` will add the migrating location to the current locality, thereby cutting it from the rest of the world.

We remark that the sublocalities of the failing one are still active, but they cannot communicate with the outside world. This translation strongly relies on the interception of routed messages by controller processes.

5. RELATED WORK

Several distributed process calculi have been proposed in the recent years, but they all have shortcomings as distributed *programming* models:

- Ambient calculi, such as the original Mobile Ambients [4] and the subsequent variants (Safe Ambients [10], Safe Ambients with Passwords [13], Boxed Ambients [2], Controlled Ambients [22]), provide a simple model of hierarchical localities with fine-grained control over locality moves and communications, but their basic mobility primitives (the `in` and `out` capabilities) require a 3-party atomic handshake which makes them costly to implement in a distributed setting as illustrated by the implementation of Mobile Ambients in the Distributed Join calculus [8].[1]

- Higher-order process calculi such as as Facile/CHOCS and D$\pi\lambda$ [25] model process mobility via higher-order communication and remote process execution, but lack an explicit notion of locality to account for potential failures or to provide a basis for access control. Furthermore, they do not allow for a running process to be migrated to a different locality, unless the process has been explicitly defined to allow for such a migration.

- Variants of the first-order asynchronous π-calculus with explicit localities such as the Distributed Join calculus [6, 11], Nomadic Pict [24], DiTyCo [12], or the π_{1l}-calculus [1], feature process migration primitives (`go` in the Distributed Join calculus, `spawn` in the π_{1l}-calculus, `migrate` in Nomadic Pict) but lack sufficient control over resource access and process migration.

Compared to these works, the M-calculus has several distinguishing features.

Its notion of programmable locality allows the definition of different forms of locality within the same calculus. In contrast, the calculi above, or even calculi such as Klaim [14, 15] (that uses generative communication as its basic form of communication), or ATF [5] (that considers distributed 2-phase transactions as processes), consider only a single form of locality.

[1]Recent work on a distributed abstract machine for Safe Ambients [17], just reinforces this point. The distributed implementation of ambients proposed there does away with the problem by implementing the `in` and `out` capabilities locally (by using cocapabilities and single-threadedness), and interpreting the `open` capability as a move to the implicit location of the parent ambient. In this interpretation, ambients no longer characterize the physical distribution of a computation, which defeats the original intent of the calculus. Furthermore, work on Boxed Ambients successfully argues against the `open` capability.

In distributed calculi, many alternatives exist when it comes to combine communication and localities. At one extreme lies the fully transparent communication of the Distributed Join calculus, where messages are routed directly to the target locality. At the other extreme lies Mobile Ambients, where communication is purely local to an ambient and remote communication must use migration primitives and explicitly encoded routes to deliver an ambient message. The Seal calculus [23] and Boxed Ambients lie between these two extremes by providing the ability to communicate across one locality boundary. In the M-calculus, we still provide transparent routing but allow messages to be intercepted each time they cross a locality boundary.

Finally, one may remark that there are striking similarities between controlling migration on one hand, and controlling communication on the other hand. For this reason, we take the further step to merge the two aspects, by considering a higher-order calculus. In our setting, migration becomes communication of a *thunk* or passivated process.

With respect to the type system, Yoshida and Hennessy's work on D$\pi\lambda$, a higher-order distributed process calculus [26], is closest to our own. D$\pi\lambda$ allows the communication of processes as thunks. In this regard, it is similar to the M-calculus. There are however several important differences that we now detail. The two calculi take a different approach to the determinacy of communication. As in the Join calculus, the D$\pi\lambda$ calculus adopts a "channel locality" invariant, which ensures a channel or resource is only present in one locality. In the M-calculus, resources of the same name can be present in different localities in order to allow for dynamic binding. The M-calculus therefore relies on addressed resources, where a resource name is annotated with a locality name, to ensure the determinacy of message routing. The "unicity of locality names" invariant in the M-calculus is more complex to ensure than the "channel locality" invariant in the D$\pi\lambda$ calculus. This is due to the passivation operator in the M-calculus. This operator has no counterpart in the D$\pi\lambda$ calculus, where the only way to obtain a thunk (frozen process) is by either specifying it, or by receiving it. The ability to passivate an active process, as in the M-calculus, is powerful but it makes the type system more complex, as the type of the passivation operator depends on some type information of the passivated process. As in the D$\pi\lambda$ calculus, a process in the M-calculus has a type that gives information on its behavior (its interface for the D$\pi\lambda$, its active localities for the M-calculus). More precisely, the type of a term is a conservative approximation of the value or process it may become. Thus locality errors in D$\pi\lambda$ and multiple locality definitions in the M-calculus only lead to type errors if the faulty process may eventually become active. Both systems also have a form of dependent types, but they are dealt with differently. In the D$\pi\lambda$-calculus, name variables can occur in types and can be bound in types, yielding dependent types. In the M-calculus, the dependency between input and output of a resource (channel) is represented as a type variable (which may stand for a locality name) that is generalized. According to the type of the argument, the type variables of the type scheme of a resource are instantiated to the correct values, and the type of a message is thus dependent on the type of the argument. In order to simplify the type system of the M-calculus, we allow cell names in the types but not name variables. We use instead name type variables, which provide us with a cleaner distinction between types and names. The advantage of dependent types appears when considering partial application: if the result of an application is a function that has a dependent type, this function may still be applied to several frozen processes having different types, which is impossible when relying on first order polymorphism. However, previous work on the typing of the Join calculus and our experience with programming in JoCaml showed us that first order polymorphism is useful, well-understood, and powerful enough when using Join patterns. Finally, both the $D\pi\lambda$ calculus and the M-calculus allow input on channels that were received previously, but the $D\pi\lambda$ uses finer input-output types for channels that could be adapted to our setting.

6. CONCLUSION AND FUTURE WORK

We have presented in this paper the M-calculus, a higher-order distributed process calculus, and an associated type system that statically enforces the unicity of locality names, a crucial invariant for the determinacy of final destination for remote messages. The M-calculus constitutes a non-trivial and powerful extension of the Join calculus [6, 11] with call-by-value higher-order functions, programmable localities, and dynamic binding.

The possibility to define, within the same calculus, different kinds of localities, with non-trivial behavior, is an important requirement for a realistic foundation of distributed mobile programming. While the M-calculus still falls short of meeting key requirements for advanced distributed programming (e.g. support for transactional behavior), we believe it constitutes an important step.

To validate the implementable character of the M-calculus, we have defined and implemented a distributed abstract machine for it [9], and, in order to prove the correctness of our abstract machine, we are working on an annotated lower-level calculus that makes the routing and passivation reduction rules more local. In parallel, we are defining notions of *observables* in order to compare, using operational equivalences, the annotated calculus to the original calculus, and to study the correctness of the translations from distributed calculi to the M-calculus.

7. REFERENCES

[1] R. Amadio. An asynchronous model of locality, failure, and process mobility. Technical report, INRIA Research Report RR-3109, INRIA Sophia-Antipolis, France, 1997.

[2] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, 2001.

[3] L. Cardelli. Types for mobile ambients. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1999.

[4] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.

[5] D. Duggan. Atomic failure in wide-area computation. In *Proceedings 4th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), S. Smith and C. Talcott eds.* Kluwer, 2000.

[6] C. Fournet. *The Join-Calculus*. PhD thesis, Ecole Polytechnique, 1998.

[7] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *In Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*. Springer Verlag, 1996.

[8] C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*. Springer, 2000.

[9] F. Germain, M. Lacoste, and J.B. Stefani. An abstract machine for a higher-order distributed process calculus. In *Proceedings of the EACTS Workshop on Foundations of Wide Area Network Computing (F-WAN)*, July 2002.

[10] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, 2000.

[11] J.J. Levy. Some results on the join-calculus. In *3rd International Symposium on Theoretical spects of Computer Software (TACS), Lecture Notes in Computer Science no 1281*. Springer, 1997.

[12] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: An Experiment in Code Mobility from the Realm of Process Calculi. In *Proceedings 5th Mobile Object Systems Workshop (MOS'99)*, 1999.

[13] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *29th ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon, 16-18 January*, 2002.

[14] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering, Vol. 24, no 5*, 1998.

[15] R. De Nicola, G.L. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science, Vol. 240, no 1*, 2000.

[16] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[17] D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2001.

[18] A. Schmitt and J.B. Stefani. The M-calculus: A Higher Order Distributed Process Calculus. Draft of the long version, available at `http://pauillac.inria.fr/~aschmitt/publications.html`, 2002.

[19] Alan Schmitt. *Conception et Implmentation de Calculs d'Agents Mobiles*. PhD thesis, Ecole Polytechnique, 2002.

[20] Alan Schmitt. Safe Dynamic Binding in the Join Calculus. In *IFIP TCS'02*, Montreal, Canada, 2002.

[21] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Box-π, wrappers and causality types. *Journal of Computer Security*, 2000. Invited submission for a CSFW00 special issue. To appear.

[22] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proceedings CONCUR 02*, 2002.

[23] J. Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Proceedings Workshop on Internet Programming Languages, Chicago, Illinois, USA*, 1998.

[24] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.

[25] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher-order processes. In *Proceedings CONCUR 99, Lecture Notes in Computer Science no 1664*. Springer, 1999.

[26] N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.