Electronic Notes in Theoretical Computer Science 66 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 25 pages

An Abstract Machine for a Higher-Order Distributed Process Calculus [1]

Florence Germain [a], Marc Lacoste [a] and Jean-Bernard Stefani [b]

[a] *France Telecom R & D,*
28, Chemin du Vieux Chêne, 38243 Meylan Cedex, France
E-mail: {florence.germain, marc.lacoste}@rd.francetelecom.com

[b] *INRIA Rhône-Alpes,*
655, Avenue de l'Europe, 38334 Saint-Ismier Cedex, France
E-mail: jean-bernard.stefani@inrialpes.fr

Abstract

This paper presents the formal specification of an abstract machine for the M-calculus, a new distributed process calculus. The M-calculus can be understood as an extension of the Join calculus that realizes an original combination of the following features: programmable localities, higher-order functions and processes, process mobility, and dynamic binding. Our abstract machine covers these different features and presents a modular structure that clearly separates the sequential (functional) evaluation mechanism from the execution core, and the latter from basic marshalling, location and routing mechanisms.

1 Introduction

The quest for a formal model for mobile and distributed programming is still on, as can be seen from the recent flurry of distributed process calculi. Key insights for these calculi are those laid out by L. Cardelli in [3], where it is argued that WAN-based distributed programming is substantially different from distributed programming in a LAN setting, and requires the introduction of different forms of barriers or localities. Notions of localities are thus present in several recent process calculi such as the Join calculus [6,9], the Seal calculus [17], Nomadic Pict [18,19], the D$\pi\lambda$ calculus [20], DiTyCo [10,11], Klaim [13], and the different variants of Mobile Ambients [4] such as Safe Ambients [8], or Boxed Ambients [2].

Each one of these calculi introduces a specific notion of locality which is characterized by its interaction protocol. For instance, in the Join calculus, the locality interaction protocol allows remote communication between localities and locality migration by means of the go construct. In Mobile Ambients, the interaction protocol corresponds to the in, out and open capabilities, which allow ambients

[1] This work was supported by project MIKADO (IST-2001-32222).

©2002 Published by Elsevier Science B. V.

to move in and out of other ambients and to dissolve an ambient boundary, respectively. Interaction protocols for localities aim to capture the different aspects of computation in a distributed system: resource access, process mobility, access control, failure modes, etc.

In a real distributed system, however, one is likely to encounter a combination of localities of different forms. It would thus be highly desirable to be able to combine, *within the same calculus*, different forms of localities. For instance, it should be possible to combine a failure model for a locality, together with an access control model. This is precisely the aim of the M-calculus [16] which introduces the idea of *programmable localities* called *cells*. In the M-calculus, each cell comprises two parts: a *membrane* that is responsible for implementing the locality interaction protocol and a *content* that executes inside a cell. As in the Join calculus and in Ambients, we give names to localities and we write $a(P)[Q]$ for the cell named a with membrane P and content Q. Since the cell membrane is an M-calculus process, we can define, within the calculus, the interaction protocol that governs a given locality.

In distributed calculi, many alternatives also exist when it comes to combining communication and localities. One extreme is the fully transparent communication model of the Join calculus, where messages are routed directly to the target locality. On the other end of the spectrum, one finds 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. Seal and Boxed Ambients lie between these two extremes by providing the ability to communicate across one locality boundary. The M-calculus offers a form of trade-off between these possibilities. We still provide transparent asynchronous interaction but communication is not direct. It consists in elementary steps that only cross a single cell membrane at a time, using a silent routing mechanism that allows each membrane to filter incoming and outgoing messages. One may remark that there are striking similarities between controlling migration and controlling communication. For this reason, to merge these two aspects, we take in the M-calculus the communication-oriented approach by considering a higher-order calculus: in our setting, migration becomes communication of a *thunk* or passivated process.

Practical distributed programming also requires some form of dynamic binding, for the exact resource to which a particular name in a program (process) should be bound need not be known until run-time and will in general be dependent on the location in which that program will execute. Thus, the M-calculus supports a form of dynamic binding, whereby the same resource names may give access to different resources in different cells.

This combination of features: programmable localities, higher-order processes, transparent but controllable [2] asynchronous communication, and dynamic binding,

[2] For general purpose distributed programming, the calculus exhibits a default behavior for localities, which allows them to be used transparently. However, specific programmer needs may require richer communication semantics. Thus, the calculus also enables this default behavior to be customized, allowing explicitly to program the locality interaction protocol.

make the M-calculus an interesting candidate for a realistic distributed and mobile programming model. However, to qualify as a realistic programming model, the M-calculus should be efficiently implementable. In this paper, we report on a preliminary implementation of the M-calculus, one that directly realizes a distributed abstract machine for the calculus. The paper details the formal specification of this abstract machine, showing that the combination of higher-order features, distribution and mobility can be effectively implemented in a distributed setting. The implementation is also succinctly and accurately described. This work shows that, even though the M-calculus constitutes a non-trivial, higher-order extension of the (distributed) Join calculus, the calculus remains implementable, and that a distributed abstract machine for it is in fact no more complex than an equivalent machine for the Join calculus.

The distributed abstract machine described in this paper is interesting in its own right. Its modular structure reflects the operational semantics of the calculus. It separates the standard sequential functional evaluation part from the rest of the computation, and identifies the key constructs required for the distributed routing of messages in presence of mobility.

The paper is organized as follows. Section 2 briefly introduces the M-calculus and its operational semantics. Section 3 formally specifies an abstract machine for the M-calculus. Section 4 describes a concrete implementation of the abstract machine in a centralized setting. Section 5 concludes the paper with a discussion of related work and future research.

2 An Overview of The M-calculus

2.1 Syntax

The syntax of the M-calculus, given in Figures 1 and 2, is a mix of call-by-value λ-calculus and of Join calculus. From the former, we have λ-abstractions and functional application. From the latter we have resource definitions with join patterns, and parallel composition. To this, we add the cell construct $a(P)[Q]$, which extends the Join calculus locality construct, together with the `pass` passivation operator, a conditional test with simple name pattern matching $[\mu = V]P, Q$, and the π-calculus restriction $\nu n.P$.

Communication in the M-calculus takes the form of an asynchronous, point-to-point exchange of messages, reflecting the dominant mode of communication in current large scale networks. Messages take the form of applications and can be of two kinds:

- Messages of the form $r\boldsymbol{V}$, where r is a resource name (the message *target*) and $\boldsymbol{V}$ is a tuple of values (the message *arguments*), are *local messages* that never cross cell boundaries. They are used for communication between processes within a given cell.
- Messages of the form $d.r\boldsymbol{V}$, where d denotes the target cell and r denotes a resource, can be used for communication between different cells. If d is of the

Syntax:

$$
\begin{array}{lll}
\mathcal{S} ::= & & \text{[system]} \\
& \epsilon[P] & \text{[root cell]} \\
| & \nu n.\mathcal{S} & \text{[restriction]} \\
\\
P ::= & & \text{[process]} \\
& \mathbf{0} & \text{[null process]} \\
| & V & \text{[value]} \\
| & \mathbf{a}(P)[P] & \text{[cell]} \\
| & (P \mid P) & \text{[parallel composition]} \\
| & (PP) & \text{[application]} \\
| & \nu n.P & \text{[restriction]} \\
| & ([\mu = V]P, P) & \text{[conditional]} \\
| & \langle D \rangle & \text{[definition]} \\
| & \mathtt{pass}_{\mathbf{a}}\ V & \text{[passivation]}
\end{array}
\qquad
\begin{array}{lll}
V ::= & & \text{[value]} \\
& () & \text{[null value]} \\
| & u & \text{[name]} \\
| & d & \text{[target]} \\
| & (V, \ldots, V) & \text{[tuple]} \\
| & \lambda x.P & \text{[abstraction]} \\
\\
D ::= & & \text{[definition]} \\
& \top & \text{[empty definition]} \\
| & J \triangleright P & \text{[reaction rule]} \\
| & D; D & \text{[composition]} \\
\\
J ::= & & \text{[join pattern]} \\
& r\boldsymbol{x} & \text{[message pattern]} \\
| & J \mid J & \text{[synchronization]}
\end{array}
$$

Names:

$$
\begin{array}{lll}
n ::= & & \text{[resolved name]} \\
& r & \text{[resource name]} \\
| & a & \text{[cell name]} \\
\\
\mathbf{r} ::= & & \text{[variable resource name]} \\
& r & \text{[resource name]} \\
| & x & \text{[variable]}
\end{array}
\qquad
\begin{array}{lll}
\mathbf{a} ::= & & \text{[variable cell name]} \\
& a & \text{[cell name]} \\
| & x & \text{[variable]} \\
\\
d ::= & & \text{[target name]} \\
& \overline{\mathbf{a}} & \text{[target membrane]} \\
| & \underline{\mathbf{a}} & \text{[target content]}
\end{array}
\qquad
\begin{array}{lll}
u ::= & & \text{[name]} \\
& a & \text{[cell name]} \\
& \mathbf{r} & \text{[variable resource name]} \\
| & d.\mathbf{r} & \text{[located resource]} \\
\\
\mu ::= & & \text{[name pattern]} \\
& u & \text{[name]} \\
| & _ & \text{[any name]} \\
| & d._ & \text{[resource at } d\text{]} \\
| & _.\mathbf{r} & \text{[located } r \text{ resource]}
\end{array}
$$

Fig. 1. The M-calculus: Syntax and Names

form $\overline{a}$, (resp. $\underline{a}$) the target resource should be located within the membrane (resp. content) of the cell named a. These *addressed messages* are routed in the calculus from cell to cell.

The different forms of names which are used in the M-calculus are gathered in Figure 1. We postulate an infinite denumerable set of *cell names*, CELL (with a distinguished element ϵ), an infinite denumerable set of *resource names*, REF (with two distinguished elements, **i**, and **o**), and an infinite denumerable set of *variables*, VAR. We let x range over VAR, r range over REF, and a range over CELL.

Values in the M-calculus can be either the null value $()$, a name u (which can be a cell name, a resource name, or an addressed resource name), a tuple of values $\boldsymbol{V}$, or a λ-abstraction. Note that it is not possible to abstract over resource names appearing in definitions: $\lambda x.\langle x \triangleright P\rangle$ is not a licit M-calculus term.

The $\mathtt{pass}_a$ construct, together with the cell construct $a(P)[Q]$, is specific to the M-calculus. It allows the membrane process P of a cell to freeze the execution

Free names:

$fn(\mathbf{a}) = fn(\overline{\mathbf{a}}) = fn(\underline{\mathbf{a}}) = \{\mathbf{a}\}$ $\quad fn(\mathbf{r}) = \{\mathbf{r}\}$

$fn(d.\mathbf{r}) = fn(d) \cup \{\mathbf{r}\}$ $\quad fn(_) = \varnothing$

$fn(d._) = fn(d)$ $\quad fn(_.\{\mathbf{r}\}) = \{\mathbf{r}\}$

$fn(()) = fn(\mathbf{0}) = \varnothing$ $\quad fn(\lambda x.P) = fn(P) \setminus \{x\}$

$fn(\nu n.P) = fn(P) \setminus \{n\}$ $\quad fn(PQ) = fn(P) \cup fn(Q)$

$fn(\langle D \rangle) = fn(D)$ $\quad fn(D; D') = fn(D) \cup fn(D')$

$fn(\top) = \varnothing$ $\quad fn(J \triangleright P) = (fn(P) \setminus fn(J)) \cup dn(J)$

$fn(r_1 \boldsymbol{x}_1 \mid \ldots \mid r_q \boldsymbol{x}_q) = fn(r_1 \boldsymbol{x}_1) \cup \ldots \cup fn(r_q \boldsymbol{x}_q)$ $\quad fn(r(x_1, \ldots, x_p)) = \{r, x_1, \ldots, x_p\}$

$fn(a(P)[Q]) = \{a\} \cup fn(P) \cup fn(Q)$ $\quad fn(P \mid Q) = fn(P) \cup fn(Q)$

$fn([\mu = V]P, Q) = fn(\mu) \cup fn(P) \cup fn(Q) \cup fn(V)$ $\quad fn(\mathtt{pass}_{\mathbf{a}}\ V) = \{\mathbf{a}\} \cup fn(V)$

$fn(\nu n.\mathcal{S}) = fn(\mathcal{S}) \setminus \{n\}$ $\quad fn(\epsilon[P]) = fn(P)$

$fn((V_1, \ldots, V_p)) = fn(V_1) \cup \ldots fn(V_p)$

Defined local names:

$dln(\nu n.P) = dln(P) \setminus \{n\}$ $\quad dln(PQ) = \varnothing$ $\quad dln(\langle D \rangle) = dln(D)$

$dln(D; D') = dln(D) \cup dln(D')$ $\quad dln(\top) = \varnothing$ $\quad dln(J \triangleright P) = dn(J)$

$dln(\mathbf{a}(P)[Q]) = \varnothing$ $\quad dln(P \mid Q) = dln(P) \cup dln(Q)$ $\quad dln([\mu = V]P, Q) = \varnothing$

$dln(\mathtt{pass}_{\mathbf{a}}\ V) = \varnothing$ $\quad dln(\mathcal{S}) = \varnothing$ $\quad dln(V) = \varnothing$

$dln(\mathbf{0}) = \varnothing$

Active cells:

$cells(\nu n.P) = cells(P) \setminus \{n\}$ $\quad cells(PQ) = \varnothing$

$cells(\langle D \rangle) = \varnothing$ $\quad cells(a(P)[Q]) = \{a\} \cup cells(P) \cup cells(Q)$

$cells(P \mid Q) = cells(P) \cup cells(Q)$ $\quad cells([\mu = V]P, Q) = \varnothing$

$cells(\mathtt{pass}_{\mathbf{a}}\ V) = \varnothing$ $\quad cells(\nu n.\mathcal{S}) = cells(\mathcal{S}) \setminus \{n\}$

$cells(\epsilon[P]) = cells(P)$ $\quad cells(V) = \varnothing$

$cells(\mathbf{0}) = \varnothing$ $\quad cells(x(P)[Q]) = \varnothing$

Fig. 2. The M-calculus: Free Names, Defined Local Names, and Active Cells

of the entire cell a. The argument of the pass operator is a function that takes two arguments : a thunk corresponding to the membrane of the passivated cell and a thunk corresponding to the content of the passivated cell. This construct, together with higher-order features, is responsible for the expressive power of the M-calculus as illustrated by the examples found at the end of the section.

An M-context is a term built according to the same grammar than for standard M-calculus terms, plus a constant $\cdot$, the hole. We use $P\{\cdot\}$ to denote M-contexts. Filling the hole in $P\{\cdot\}$ with an M-calculus term Q results in an M-calculus term noted $P\{Q\}$. Evaluation contexts in the M-calculus are M-contexts built according to the following grammar:

$$\mathbf{E} ::= \ \cdot \ \mid \ \mathbf{E}V \ \mid \ P\mathbf{E} \ \mid \ \nu n.\mathbf{E} \ \mid \ (\mathbf{E} \mid P) \ \mid \ a(P)[\mathbf{E}] \ \mid \ a(\mathbf{E})[P] \ \mid \ \epsilon[\mathbf{E}]$$

We use the following notational conventions: we write $fn(P)$ for the set of free names of P, and $dn(J)$ for the set of defined names of a join pattern J. More precisely, $dn(r_1 \boldsymbol{x}_1 \mid \ldots \mid r_q \boldsymbol{x}_q) = \{r_1, \ldots, r_q\}$. We also write $dln(P)$ for the set of

defined local names of P, and *cells*(P) for the multiset of the names of active cells found in P. These sets are defined recursively in Figure 2.

We note $\theta = \{t_1/u_1, \ldots, t_q/u_q\}$ a finite substitution where the term t_i is substituted to the name u_i. We note $P\theta$ or $P\{t_1/u_1, \ldots, t_q/u_q\}$ the image of the term P under substitution θ. We use $\boldsymbol{t}$ to denote finite tuples of terms $(t_1, \ldots, t_q)$. When tuples $\boldsymbol{t}$ and $\boldsymbol{u}$ are of the same size, we note $\{\boldsymbol{t}/\boldsymbol{u}\}$ the substitution $\{t_1/u_1, \ldots, t_p/u_p\}$. The same notational convention applies to tuples of tuples. We also make use of the notation $\lambda.P$ to stand for a *thunk* $\lambda x.P$, with x not free in P. 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 $r_1\boldsymbol{x_1} \mid \ldots \mid r_q\boldsymbol{x_q} \triangleright P$, the names and variables n, x, and x_{ij} are bound in P.

2.2 Operational Semantics

The operational semantics of the M-calculus is defined in the CHAM style [1], using a structural equivalence, $\equiv$, and a reduction relation, $\rightarrow$. The structural equivalence, $\equiv$, is the smallest equivalence relation that satisfies the rules S.NU.PAR to S.CONTEXT given in Figure 3, where the parallel composition operator "|" for processes, and the composition operator ";" for definitions are taken to be commutative and associative, with $\mathbf{0}$ and $\top$ as their neutral elements, respectively. The structural rules comprise scope extrusion rules for the restriction operator and standard rules for equivalence under α-conversion and congruence for evaluation contexts.

The reduction relation for the M-calculus, $\rightarrow$, is defined as the smallest relation that satisfies the rules given in Figure 3. The computing rules for the conditional branch make use of the *match*() predicate defined by:

$$match(_, V) \qquad match(u, u) \qquad match(\mathbf{a}._, \mathbf{a}.\mathbf{r}) \qquad match(_.\mathbf{r}, \mathbf{a}.\mathbf{r})$$

$$match(_._, \mathbf{a}.\mathbf{r})$$

Rule R.COMM is similar to the JOIN rule of the Join calculus: when a set of local messages matches the pattern of a definition, a new instance of the guarded process in the definition is triggered. Rule R.PASSIV is novel and defines the semantics of the `pass` construct. An instruction $\mathtt{pass}_a\ V$ can only be activated within the membrane of a cell named a. Its effect is to freeze the membrane process and the content process of the cell and to pass the resulting thunks as arguments to function V.

The routing rules deal with the transport of messages to target resources. By definition, routing of local messages only takes place between membrane and content within the same cell. In routing rules for addressed messages, we write $\underline{\overline{b}}$ for either $\overline{b}$ or $\underline{b}$. These rules handle the different cases of messages currently found outside their target cell, within a cell membrane or within a cell content. Messages which have reached their target cell membrane or content are just turned into local messages (rules R.A$_1$.MM, CM, EM, CC, MC). Messages outside a given cell that target resources located within subcells or in the content of the target cell are filtered by the cell membrane on the $\mathbf{i}$ port (rules R.A$_1$.EC and R.A$_2$.IN). Messages

within a cell membrane can flow freely inside the cell content or outside the cell (rules R.A$_2$.MC, ME). Finally, messages inside a cell content targeting subcells in the membrane or cells outside of the current cell are filtered by the cell membrane on the $\mathbf{o}$ port (rule R.A$_2$.OUT). The routing rules in the M-calculus correspond to elementary routing steps, that enforce the two following principles: all messages are routed one boundary at a time (exterior to membrane, membrane to content, and conversely); no message can enter or leave the content of a cell without having been handled first by the membrane of the cell.

2.3 Programming in the M-calculus

The following examples illustrate the main constructs of the calculus.

EXAMPLE 1. Transparent communications à la Join calculus can be implemented in the M-calculus through the use of forwarders in cell membranes. A sample forwarder is given by the following process:

$$Fwd = \langle \mathbf{i}(d,r,v) \triangleright d.r\ v\ ;\ \ \mathbf{o}(d,r,v) \triangleright d.r\ v \rangle$$

Placed in a cell membrane, Fwd just releases incoming and outgoing messages in the membrane. The routing rules ensure that the released messages continue their journey towards their target cell. Such a forwarder is completely stateless and works even if the target cell moves while the message is being routed to it.

EXAMPLE 2. Cell mobility can be implemented using higher-order messages and passivation (cf the `go` construct in the Join calculus and the `in` and `out` capabilities in Mobile Ambients). Cell $Q^m(a)$ below can be moved to a different cell:

$$Q^m(a) = a(Fwd \mid \langle \mathsf{go}\, u \triangleright Go(a,u) \rangle)[Q]$$
$$Go(a,u) = \mathsf{pass}_a\ \lambda p\, q.(\overline{u}.\mathsf{enter}\ \lambda.a(p())[q()])$$

A $(\mathsf{go}\, u)$ request results in the passivation of the cell and its sending as a thunk to the cell named u. If the request comes from the outside of the cell, the result is an objective form of move. If the request comes from the content of the cell, the result is a subjective form of move. The membrane of cell u can contain the process $Enter(u)$ below to allow the insertion of a new cell in its content:

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

EXAMPLE 3. Cell membranes can implement various forms of control. Cell $Q^o(a)$ below can be suspended, resumed, dissolved (cf the `open` capability of ambients), and updated with a new membrane process (we note simply r a message of the form $r()$):

$$\begin{aligned}
Q^o(a) &= \nu s\, on.a\ (Fwd \mid \langle\ \mathsf{suspend} \mid on \triangleright S(a,s);\ \mathsf{resume} \mid s\, f \triangleright R(a,f,on);\\
&\qquad \mathsf{open} \triangleright O(a);\ \mathsf{update}\ f \triangleright U(a,f)\ \ \rangle\ )[Q]\\
S(a,s) &= \mathsf{pass}_a\ \lambda p\, q.a(p() \mid (s\, q))[\mathbf{0}]\\
R(a,f,on) &= \mathsf{pass}_a\ \lambda p\, q.a(p() \mid on)[f()]\\
O(a) &= \mathsf{pass}_a\ \lambda p\, q.q()\\
U(a,f) &= \mathsf{pass}_a\ \lambda p\, q.a(f())[q()]
\end{aligned}$$

Structural Equivalence Rules:

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

S.NU.TOP $$\frac{}{\epsilon[\nu n.P] \equiv \nu n.\epsilon[P]}$$

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

S.α $$\frac{P =_\alpha Q}{P \equiv Q}$$

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

S.CONTEXT $$\frac{P \equiv Q}{\mathbf{E}\{P\} \equiv \mathbf{E}\{Q\}}$$

Reduction Rules: Computing

R.BETA $$\frac{}{(\lambda x.P)V \to P\{^V/_x\}}$$

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

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

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

R.COMM $$\frac{\langle D\rangle = \langle D_0\ ;\ r_1\boldsymbol{x_1} \mid \ldots \mid r_n\boldsymbol{x_n} \triangleright P\rangle}{\langle D\rangle \mid r_1\boldsymbol{V_1} \mid \ldots \mid r_n\boldsymbol{V_n} \to \langle D\rangle \mid P\{^{\boldsymbol{V_i}}/_{\boldsymbol{x_i}}\}}$$

R.P.EQUIV $$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

R.S.EQUIV $$\frac{\mathcal{S}_1 \equiv \mathcal{S}_1' \qquad \mathcal{S}_1' \to \mathcal{S}_2' \qquad \mathcal{S}_2' \equiv \mathcal{S}_2}{\mathcal{S}_1 \to \mathcal{S}_2}$$

Reduction Rules: Routing Local Messages

R.L.MC $$\frac{r \notin dln(P) \qquad r \in dln(Q)}{a(P \mid r\boldsymbol{V})[Q] \to a(P)[Q \mid r\boldsymbol{V}]}$$

R.L.CM $$\frac{r \in dln(P) \qquad r \notin dln(Q)}{a(P)[Q \mid r\boldsymbol{V}] \to a(P \mid r\boldsymbol{V})[Q]}$$

Reduction Rules: Routing Addressed Messages

R.A$_1$.MM $$\frac{r \in dln(P)}{a(P \mid \overline{a}.r\boldsymbol{V})[Q] \to a(P \mid r\boldsymbol{V})[Q]}$$

R.A$_1$.EM $$\frac{r \in dln(P)}{\overline{a}.r\boldsymbol{V} \mid a(P)[Q] \to a(P \mid r\boldsymbol{V})[Q]}$$

R.A$_1$.CM $$\frac{r \in dln(P)}{a(P)[Q \mid \overline{a}.r\boldsymbol{V}] \to a(P \mid r\boldsymbol{V})[Q]}$$

R.A$_1$.CC $$\frac{r \in dln(Q)}{a(P)[Q \mid \underline{a}.r\boldsymbol{V}] \to a(P)[Q \mid r\boldsymbol{V}]}$$

R.A$_1$.MC $$\frac{r \in dln(Q)}{a(P \mid \underline{a}.r\boldsymbol{V})[Q] \to a(P)[Q \mid r\boldsymbol{V}]}$$

R.A$_1$.EC $$\frac{r \in dln(Q)}{\underline{a}.r\boldsymbol{V} \mid a(P)[Q] \to a(P \mid \mathbf{i}(\underline{a}, r, \boldsymbol{V}))[Q]}$$

R.A$_2$.IN $$\frac{b \in cells(P) \cup cells(Q) \qquad b \neq a}{\overline{\underline{b}}.r\boldsymbol{V} \mid a(P)[Q] \to a(P \mid \mathbf{i}(\overline{\underline{b}}, r, \boldsymbol{V}))[Q]}$$

R.A$_2$.MC $$\frac{b \in cells(Q) \setminus cells(P) \qquad b \neq a}{a(P \mid \overline{\underline{b}}.r\boldsymbol{V})[Q] \to a(P)[Q \mid \overline{\underline{b}}.r\boldsymbol{V}]}$$

R.A$_2$.ME $$\frac{b \notin cells(P) \cup cells(Q) \qquad b \neq a}{a(P \mid \overline{\underline{b}}.r\boldsymbol{V})[Q] \to a(P)[Q] \mid \overline{\underline{b}}.r\boldsymbol{V}}$$

R.A$_2$.OUT $$\frac{b \notin cells(Q) \qquad b \neq a}{a(P)[Q \mid \overline{\underline{b}}.r\boldsymbol{V}] \to a(P \mid \mathbf{o}(\overline{\underline{b}}, r, \boldsymbol{V}))[Q]}$$

Fig. 3. Structural Equivalence and Reduction Rules

3 The M-calculus Abstract Machine

We now describe a distributed abstract machine (AM) for the M-calculus called CLAM (CeLular Abstract Machine).

3.1 Design Principles

Validation of an M-calculus-based "design-to-implementation" formal approach to building realistic reliable distributed infrastructures meant demonstrating the effectively implementable character of the model. Thus, the M-calculus should be refined in a formal specification of an implementation realizing the model – which takes in this paper the form of an AM. The abstraction level of the AM should both closely reflect existing infrastructures for distributed computing, while still being close enough to the calculus to facilitate the expression and proof of implementation-related properties. With that perspective in mind, we made the following design decisions for the AM:

- The AM is obtained by a direct refinement of the M-calculus: the AM internals are strongly inspired by the core primitives of the calculus. Furthermore, reduction on AM states is obtained by a nearly one-to-one translation of the M-calculus reduction relation.
- The AM is effectively implementable in an asynchronous distributed setting. The AM provides a clear separation between logical and physical distribution, as in the Join calculus, where the top-level location in a location tree stands for a physical site hosting the tree. Truly distributed interaction occurs by asynchronous message passing between physical sites, i.e., between top-level locations. Thus, the corresponding AM reduction rules only involve a single location, and possibly a global ether for message sending or reception (see Figure 4 and reduction rules such as M.A_1.EC.R, M.A_2.ME.R, or M.FORWARD in Figures B.3 and B.4). In particular, routing-related decisions are purely local.
- The AM organization is modular:
  - ◇ The AM clearly separates the evaluation of functional terms, handled by a well-identified evaluator, from the execution core dedicated to concurrency and distribution management.
  - ◇ Name management and routing mechanisms are separated from the computational part of the AM. At their creation, cell names are guaranteed to be globally unique. This property is ensured in the calculus using the ν operator and scope extrusion rules, and is refined in the AM using qualified names including a reference to the location who created them. A key feature to obtain a deterministic routing algorithm for addressed messages is to preserve cell name unicity during execution. The calculus enforces this using the type system of [16] (subject reduction). At the implementation-level, cell name uniqueness is achieved by a distributed lookup service containing the current localization of definitions (in locations), and of locations (on physical units of distribution), and which serves as a basis to refine in the AM the routing discipline of the M-calculus.

3.2 Overview

We make three slight modifications to the source language:

(1) To allow a clear identification of a CLAM component specialized in the evaluation of funcional terms, we introduce the syntactic category of *expressions*, which corresponds to processes and values in the M-calculus grammar, and on which the functional evaluator will perform computations to return values.

(2) We add two special values, $\mathsf{reify}_{\mathsf{membrane}}$ and $\mathsf{reify}_{\mathsf{content}}$, respectively devoted to the reification of the membrane and of the content of a cell.

(3) In the M-calculus, a membrane may comprise a parallel composition of cells. In the sequel, we only consider a subset of the calculus where a membrane process remains *flat* throughout execution, i.e., does not contain any subcell. This restriction simplifies the description of the AM (an AM for the full calculus can easily be derived from the one we describe below), and can be enforced by a type system that is a slight variant of the type system described in [16]. This type system also enforces a property of unicity of cell names, which is crucial for guaranteeing routing determinacy in the calculus and in the abstract machine, the description of the latter being strongly based on this property.

The CLAM takes the form of a collection of *locations*, possibly distributed on several *machines*. Machines are connected by the *global ether* E, which, at any instant, contains the set of messages in transit between different machines. A machine represents the top-level unit of execution, and is typically the abstraction of a UNIX system process. To support routing, each machine contains a *lookup service* to locate both communication resources (in which location is a resource defined?) and locations (on which machine does a location reside?). We will speak of *local routing*, and of *remote routing*, respectively, depending on whether messages are routed to the same machine or to a different one. A location hosts a number of flat (i.e. non-cell) processes living together at the same level of the cell hierarchy and holds pointers to a number of children locations. Thus, locations are organized in a forest, each machine containing a single location tree.

A cell membrane is represented in the CLAM by a special kind of location called *membrane-location*. The flat processes hosted by a membrane-location are the processes that constitute the cell membrane. A membrane-location has two kinds of children locations: a single *ether-location* (or simply *ether*), which collects the flat processes found at the top level of the cell content, and subordinate membrane-locations that implement the subcells found in the cell content. Thus, each element of a location tree is either an ether-location (leaf), or a membrane-location (node), the latter possessing exactly one ether-sublocation. We call the root of each location tree a *top-location*, generally written as $l_\top$. For commodity, a virtual root location for the entire system (machines and global ether) named $\top$ is also introduced (see Figure 4).

Each location in the CLAM includes an *execution engine* and a *functional evaluator* with a well-defined interface between the two components. The execution

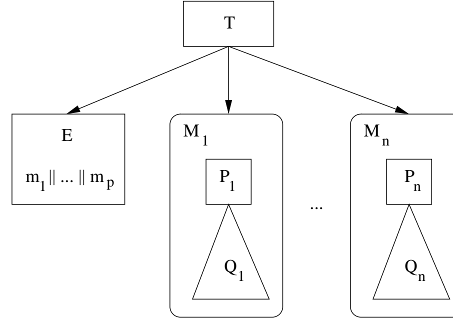

Fig. 4. The CLAM Distribution Model

engine maintains a local location state. The functional evaluator simply returns values to the execution engine, given expressions as input. More precisely, the functional evaluator operates on applications and abstractions, and returns values which are either M-calculus values, or *extended values* corresponding to the non-functional primitives of the M-calculus, namely, special values $\mathsf{reify}_{\mathsf{membrane}}$ and $\mathsf{reify}_{\mathsf{content}}$, conditional branch, cell, parallel composition, restriction, definition and passivation constructs. The formal characterization of the evaluator is omitted for brevity.

3.3 Machine and Location State

The notational conventions and syntactic categories needed for the full formal description of the CLAM and for a detailed understanding of its reduction rules are gathered in appendix A.

For implementation purposes, we enrich the set of values with a special kind of name: *internal names* are rich names used to designate internally locations, or entities coming from the calculus such as resources or cells. They hold the name of the machine they were created on, and appear typically in process environments, where calculus-level names are mapped to internal names.

A location l is described by a tuple $\langle \alpha, \mathcal{D}, \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{L} \rangle$. The *location name* l is unique at the level of the entire system. The *name* α *of the corresponding cell* in the calculus is only defined for membrane-locations. $\mathcal{D}$ denotes the set of *activable definitions* found in l, while $\mathcal{H}$ refers to the *message heap*, where messages waiting for a communication on a locally-defined name are kept. $\mathcal{R}$ denotes the location *run-queue*, where processes to be executed are stored. Finally, $\mathcal{E}$ and $\mathcal{L}$ are used to keep track of the sublocations of l, respectively of kind ether and membrane. Note that $\mathcal{E}$ is always a singleton for membrane locations, or empty for ether locations – in which case $\mathcal{E}$ is noted $\sharp$.

A machine is represented by a tuple $\langle m : N : O : L \rangle$, and is uniquely identified in the distributed system by its *machine name* m. N is a *name factory*, used to create fresh internal names. O is the machine local *lookup service*. L keeps track of locations residing on m, by mapping location names to $\langle \alpha, \mathcal{D}, \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{L} \rangle$ tuples.

3.4 Reduction Rules

The evolution of a location is described by reduction rules which either concern activity within a single machine, or a machine sending/receiving messages to/from the global ether. For instance, we write

$$m : N : O : L\{l \xmapsto[l_e:\mathcal{L}]{\mathcal{D}:\mathcal{H}} \langle \alpha, \mathcal{R} \rangle, \ldots\} \parallel E\{m \to m' \mapsto \mathcal{M}\} \longrightarrow$$
$$m : N' : O' : L'\{l \xmapsto[l_e:\mathcal{L}']{\mathcal{D}':\mathcal{H}'} \langle \alpha, \mathcal{R}' \rangle, \ldots\} \parallel E\{m \to m' \mapsto \mathcal{M} :: msg\}$$

for a machine m where: (i) the state $\langle \alpha, \mathcal{D}, \mathcal{H}, \mathcal{R}, l_e, \mathcal{L} \rangle$ of a location l becomes $\langle \alpha, \mathcal{D}', \mathcal{H}', \mathcal{R}', l_e, \mathcal{L}' \rangle$; (ii) a message msg is sent on the network (global ether E) from machine m to machine m'. In a rule context, we write any non significant component of the location state as $-$, e.g., $l \xmapsto[-]{-} \langle \alpha, \mathcal{R} \rangle$.

The core of the execution engine mainly deals with sequential computation (see Figure B.1). To guarantee a fairness of execution property, i.e., to ensure that a process located in the run-queue will eventually be run after a finite number of computational steps, newly created processes are in general placed at the end of the run queue: for instance, a parallel composition $P \mid Q$ is interpreted by running P immediately, and scheduling Q for later execution (M.PRL rule). An expression is processed by delegation to the functional evaluator. The evaluation result (an extended value) is then placed at the end of the run-queue. The M.EVAL rule in fact describes the interface between the execution engine and the functional evaluator. Two abstract encodings $[\![\cdot]\!]_{load}$ and $[\![\cdot]\!]_{unload}$ capture expression loading and unloding to and from the functional evaluator. We do not describe further the functional evaluator, but one could plug in for this purpose any evaluator for value passing λ-calculus, assuming that the range of values handled by the evaluator covers those defined in the calculus. The creation of a new resource or cell name is simply expressed by the creation of a new internal name i which is added to N (M.NEW rule). In rules M.IF.THEN and M.IF.ELSE, we use an extension of the environment ρ to name patterns and values.

Local communications are specified by rules (see Figure B.2) which resemble rules for a Join calculus AM [14], since the involved primitives are direct Join calculus constructs. In M.DEF, a newly encountered definition is stored in $\mathcal{D}$ – along with the appropriate environment. The internal names corresponding to defined names become new heap entries. Entries mapping each defined name with the current location are also added in the lookup service. The received names $rn(J)$ in a join pattern J are simply the argument variables which appear in message patterns. For a new message, if the name of the recipient is already present as a heap entry, i.e., if the target definition has already been processed, the message is consumed and its arguments are queued in the matching heap entry. Otherwise, the message is put back at the end of the run-queue for further processing (M.MSG rule). The effective activation of a definition, i.e., the firing of a join-pattern body, becomes possible when, for each defined name of the join-pattern, there is at least

one message pending in the heap (M.COMM rule).

Operations that change the structure of the forest of locations are cell creation and cell passivation/activation. The discussion of the latter is postponed until the description of the cell migration mechanism. Cell creation (M.CELL rule) occurs when the process $b(P)[Q]$ is run. A sibling location l' is created for the new cell membrane, together with a child l'' of l' for the content. The run-queues of l' and l'' are then initialized with P and Q respectively. At start-up time, a given machine only contains a top-location with a cell process in its run queue to trigger the creation of a location tree by spawning new top-location descendants: a new location is created for the new cell content, and the top-location then represents the new cell membrane in future reduction steps (M.CELL$_\top$ rule).

We now describe the other AM reduction rules, which concern remote communication and migration.

3.5 Remote Communication and Migration

Routing relies upon a distributed lookup service. Various update policies are possible for the lookup service viewed here as a distributed database, for instance, either by maintaining consistent views using atomic broadcast, or allowing some views to be temporarily inconsistent, but using forwarders to bounce messages towards the correct destination. The database should satisfy the following property:

Definition 3.1 A distributed lookup service is *sound* iff in case of temporarily inconsistent views of the distributed database, the only consequence on message routing is a delayed message emission or reception, provided inter-machine communications are reliable (the global ether does not discard messages).

In our AM, we chose a particular formalization of the distributed database where each database fragment is a local lookup service for definitions and locations. We also chose a particular routing algorithm R* which uses forwarders and piggyback routing information updates on communication messages. Timestamps are used to represent the various instants of migration of cells or definitions, in order to avoid communications inconsistencies due to stale routing information contained in the lookup service.

In R*, routing is captured by two types of rules, for local and addressed messages (see Figures B.3 and B.4). Figure 5 summarizes the movements of messages between locations, represented as diamonds, both for the local and addressed message case. On the figure are also shown cell membrane and content boundaries, which give their names to the reduction rules, according to whether a message is going from/to the exterior of a cell (E), its membrane (M), or its content (C), as in rule M.L.CM which describes the migration of a local message from cell content to membrane.

Rules for local messages closely follow the corresponding rules in the calculus. For instance, AM rule M.L.MC implements calculus rule R.L.MC, and allows messages to move down in the location tree: a message $r\widetilde{V}$ can directly enter the

ether-sublocation of the current location iff resource r is defined inside that sublocation.

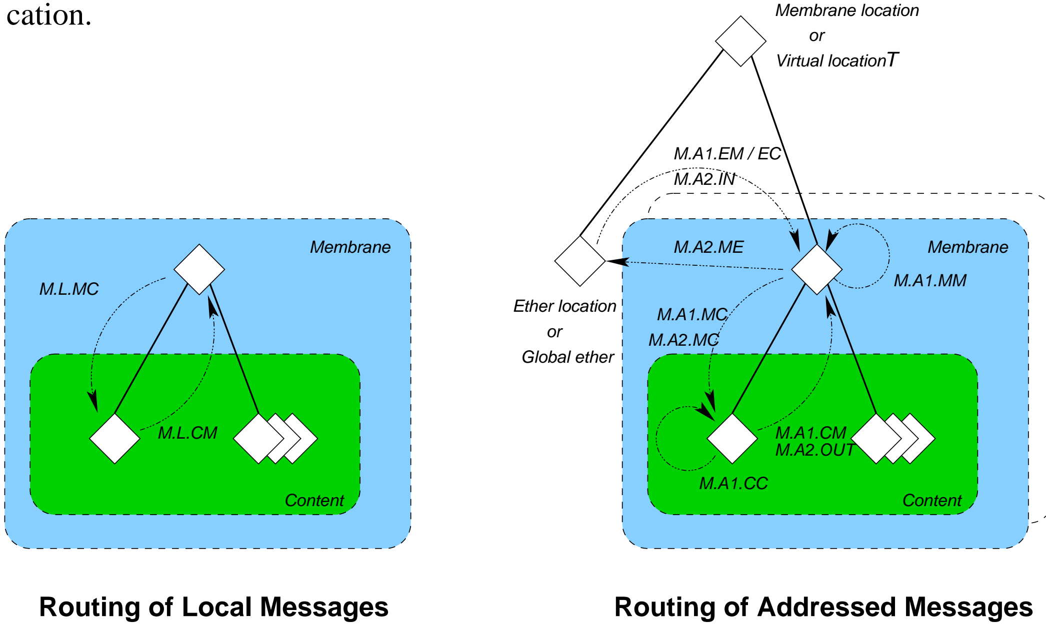

Fig. 5. Routing in the CLAM

The routing of addressed messages includes *physically-located routing* rules, e.g., M.A$_1$.EM.L, M.A$_1$.EC.L, or M.A$_2$.ME.L, and *physically-remote routing* rules, e.g., M.A$_1$.EC.R and M.A$_2$.ME.R. In both cases, the messages are routed "as is" until they reach their destination, i.e., the location which implements the target cell. Here are a few sample rules:

- **Physically-located rules:** a message targeted to a resource defined in location l, and currently located in its surrounding ether, can directly enter its destination (M.A$_1$.EM.L rule). Message routing between a membrane-location and its surrounding ether is also specified by rules M.A$_2$.IN.L and M.A$_2$.ME.L. The target cell must simply be known by the lookup service.
- **Physically-remote rules:** when a message is targeted at a physically distant location, it is packed at the top-location of the initial machine, together with routing information (current environment and state of the local lookup service), and sent over the wire towards the target machine ; additionally, forwarders are left in the local lookup service for names which occur in the message arguments (M.A$_2$.ME.R rule). When received at the top-location $l_\top$ of the target machine, the message is unpacked and added at the end of the run-queue of $l_\top$ (M.A$_1$.EC.R rule). The lookup service is updated, by merging the just received routing information with its own. These two rules are similar to the remote communication rules of the AM for the Join calculus [14]. Finally, the M.FORWARD rules deals with messages sent to a machine which does not contain the target location any more (because of a previous migration).

Cell passivation is described by rule M.PASSIV, while cell activation is described by rules M.MEMBRANE and M.CONTENT (see Figure B.2). The execu-

tion of $\mathtt{pass}_a\ V$ induces a double reification of current location l, together with its descendants, using values $\mathsf{reify}_{\mathsf{membrane}}$ and $\mathsf{reify}_{\mathsf{content}}$, which are then passed as arguments to V. The resulting process is moved to the ether-location found among the siblings of l. The passivated location subtree is removed from the set L of locations currently hosted by the current machine, using an $extract$ function, both to compute the $\mathsf{reify}_{\mathsf{membrane}}$ and $\mathsf{reify}_{\mathsf{content}}$ values from the state of the subtree, and to update L [3]. Passivation is restricted to occur inside a location which is not a top location. The M.A$_2$.ME.R rule includes a condition specifying the update of the local lookup service before the remote message sending operation. A forwarder is left in place in the local lookup service. Unlike in the Join calculus AM, the local lookup service cannot be updated with the target locations of all migrating resources: in general, a passivated resource will be activated in a new location which is unknown at passivation-time (only the target machine is known). Indeed, this target location is dynamically determined after the message reception operation has been completed on the remote machine. The idea is then to modify the local lookup service for each passivated location (resp. each passivated resource) contained in the message, so that: (i) each passivated location is mapped to the target machine ; (ii) passivated resources are mapped to a special location written $\daleth$ (*unknown location*), which is itself mapped to the target machine. A direct optimization can be made, since all passivated resources coming from a strict sublocation of the passivated location keep their own location after activation.

The M.MEMBRANE rule leads to the unpacking within the current location of the components of the $\mathsf{reify}_{\mathsf{membrane}}$ process using an $insert_{flat}$ function [4]. Reified activable definitions, and pending messages, are moved respectively in the local set of activable definitions and message heap. The reified run-queue is inserted at the end of the local run-queue. The lookup service is then updated with the new location of unpacked elements. The M.CONTENT rule leads to the unpacking within the current location of the components of the $\mathsf{reify}_{\mathsf{content}}$ process using an $insert_{sublocs}$ function. The reified ether-sublocation is directly merged with the current location, while membrane-sublocations lead to the creation of siblings locations. As for the M.MEMBRANE rule, the lookup service is also updated.

The AM satisfies a few properties, which are invariant under reductions:

Invariant 3.2 *On each machine m, locations are organized in a tree.*

Invariant 3.3 *If a machine m hosts a location l, then the name of that location is contained in the lookup service of m:*

$$\forall (m,l) \in (\text{MNAME}\times\text{LOCNM})\ \ l \in \mathsf{dom}\, m.L \Longrightarrow \exists\, t \in \text{TIMESTAMP}\ \ \{l \mapsto \langle m,t\rangle\} \subset m.O$$

Invariant 3.4 (i) *When a message targeted at a name $a.r$ is received on a machine m, either there exists on m a location named a or a forwarder for the*

[3] Given a location $l : \langle \alpha, \mathcal{D}, \mathcal{H}, \mathcal{R}, l_e, \mathcal{L}\rangle$, we use value $\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D},\mathcal{H},\mathcal{R})$ to reify the activable definitions $\mathcal{D}$, message heap $\mathcal{H}$, and run-queue $\mathcal{R}$ of a location, and value $\mathsf{reify}_{\mathsf{content}}(l_e,\mathcal{L})$ to reify all the sublocations of l, i.e., the ether l_e, the locations of $\mathcal{L}$, and all their descendants.

[4] For instance, this function can be defined by $insert_{flat}(\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}_m,\mathcal{H}_m,\mathcal{R}_m),\mathcal{D},\mathcal{H},\mathcal{R}) = \langle \mathcal{D}\cup\mathcal{D}_m, \mathcal{H}\cup\mathcal{H}_m, \mathcal{R} :: \mathcal{R}_m\rangle$.

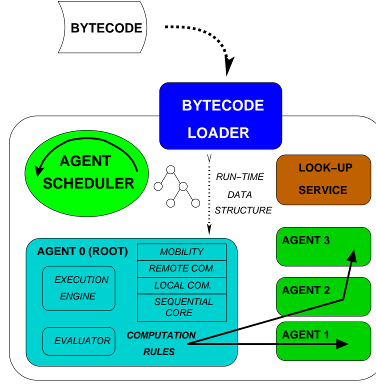

Fig. 6. Structure of the VM

message towards another machine.

(ii) *When a cell passivated as a message argument is sent from machines m to m', then a forwarder from m to m' is created locally.*

The third invariant guarantees routing determinacy between machines (a message can only be routed towards a single machine at a given time), and between locations of a same machine (in a given machine, a message can only be routed towards a single location at a given time).

Conjecture 3.5 (Soundness) *The family $(O_m)_{m \in \text{MNAME}}$ of lookup services using R* as a routing algorithm is sound.*

Finally, we deem the following property to be preserved under reduction, which may be a starting point to prove the correctness of the CLAM with respect to the M-calculus.

Conjecture 3.6 (Correctness) $\forall m \in \text{MNAME}\ \forall (l, r) \in (\text{LOCNM} \times \text{REF})$ *such that* $l \in \mathsf{dom}\, m.L \ \wedge\ l.\mathcal{R} = \ldots :: (\rho, P) :: \ldots \ \wedge\ r \in \mathsf{dom}\, \rho$:

$$l \Subset m.O(\rho(r)) \Longrightarrow r \in dln(P)$$

4 The C-VM Implementation

In this section, we briefly describe a centralized implementation of the M-calculus called C-VM (Cellular Virtual Machine), based on the CLAM defined in the previous section. The goal for the design of the C-VM platform was to obtain a portable and easily extensible implementation of the CLAM of reasonable complexity, leaving performance and other optimizations for further study.

We chose to implement an abstract machine code interpreter, both because of the need to be able to move code between machines, requiring a common executable format, and because it allows for an interactive evaluation of M-calculus expressions. Thus, the C-VM platform comes in the form of a compiler which translates M-calculus terms into bytecode, which is in turn run by a virtual machine (VM). The compiler is written in OCaml. The VM itself is written in Java. In view of building a distributed run-time, this decision makes it possible to interoperate with many existing distributed middleware written in Java, such as Java RMI.

Contrary to implementations like ZINC [7], JOCAML or TyCO [10,12] which use registers or code closures with a bytecode fairly close to assembly language, we chose to make the bytecode closer to the calculus syntax than to the underlying hardware – a feature also found in the Java Virtual Machine. This design decision is consistent with the CLAM presented in section 3 which keeps a fairly high-level view of a distributed system by masking a lot of implementation features. This choice brings several benefits: at the language level, it could make a correctness proof of the implementation easier; it is also a first step towards an efficiency-oriented run-time: as pointed out in [12], reflecting the nesting of process terms in the byte code structure allows direct optimizations: each VM manipulation of internal data structures involving process terms can be expanded into direct transformation of bytecode instruction sequences, an operation much lighter in term of execution costs.

The bytecode structure is reminiscent of [10]: language primitives are compiled into sequences of instructions, each instruction being a sequence of blocks. The first block is always the opcode, e.g, `app`, `def`, `join`, the following blocks being code pointers, e.g., offsets to the next bytecode instruction.

The VM implementation closely reflects the structure of a centralized CLAM. It consists of a *bytecode loader*, a number of *agents* running concurrently, a *scheduler*, and of a set of *services* (see Figure 6).

The heart of the VM resides in the agents which exactly mirror the CLAM formal locations. An agent is composed of an execution engine, and of an evaluator of lambda expressions whose interface mirrors the formal interface between execution engine and functional evaluator in the CLAM.

The execution engine implements the CLAM reduction rules. We use data structures such as rich names for references, or queues for processes waiting for execution or communication. Potentially activable definitions are kept in the *defspace* area which maps a definition descriptor to a definition. The descriptor includes a defspace position index, an activability status bit for the definition, and a bit array which identifies the presence of messages on each definined receiver of the join-pattern. Synchronization is implemented in the manner of JOCAML [5] by a run-time "check and update" of the definition status for each new incoming message. Processes waiting for a communication on a reference are queued in the *heap*. Internally, a reference r holds the defspace position index of the definition which defines r. The other elements of the execution engine are the *run-queue* of processes ready to be executed, the *ether* which stores messages for inter-agent communications, and a set of agent names to keep track of children locations. The mapping betwen agent names and agent pointers is kept in a centralized agent table.

Each agent is implemented by a Java thread. Scheduling is done both at the agent-level to determine the next applicable reduction rule, and at the VM-level to choose the next agent that will change state. The latter is done by a simple loop which visits all agents in the agent table. Execution terminates when no agent changed state after the last iteration. However, both scheduling policies may easily be modified.

Finally, the VM includes facilities to create fresh channel or agent names, unique at the VM-level, and a look-up service allowing, given a reference r, to locate an agent hosting a definition defining r.

5 Conclusion

We have presented the M-calculus, a new higher-order distributed process calculus, together with a supporting abstract machine. The M-calculus can be understood as a significant extension of the Join calculus, with programmable localities, higher-order processes, process mobility and dynamic binding. The associated abstract machine (CLAM), captures all these different features while remaining no more complex than an abstract machine for the Join calculus.

There is an increasing number of attempts to formalize distributed infrastructures for mobile computation. To our knowledge, the works closest to ours are Nomadic Pict [18,19], PAN [15], DiTyCO, and work on a Join calculus AM called JAM [14]. Pict provided a starting point by formalizing a concurrent implementation of the π-calculus. Pict, however, did not address distribution and mobility issues. Nomadic Pict extended Pict to build and reason about mobile agent infrastructures. There are however several important differences between the M-calculus and Nomadic Pict: Nomadic Pict is not higher-order, does not contain programmable localities, and only supports flat localities. Also, an AM for Nomadic Pict has not yet been described, although a description of high-level transparent communication primitives in terms of low-level non-transparent ones has been defined and proven correct.

Safe Ambients, TyCO, and the Join calculus each have a formally described AM. If TyCO and the JAM follow a communications-oriented approach using asynchronous π-calculus channels, on concurrent objects and definitions respectively, PAN investigates a migration-oriented approach to express all computations in terms of pure migration [4]. In the C-VM design, DiTyCO provided a starting-point for the bytecode structure to directly reflect the organization of the source program. The JAM was another direct source inspiration for our work on the CLAM. The M-calculus and the Join calculus are significantly different, however. As a result, the CLAM is very close to the JAM but differs in important aspects. In particular the migration primitives are different (remote migration versus passivation and remote communication). If the underlying calculi may differ in the structure of distributed system partitions (flat/hierarchical), in the implementation, we always find a separation between physical distribution (C-VM machines) and logical distribution (C-VM locations). Another important difference concerns whether locations are single-threaded (PAN, C-VM) or allow multiple threads to run (DiTyCO, JAM).

In the future, we plan to make our C-VM implementation more efficient and to extend it to a fully distributed one. We also intend to work on a correctness proof for the CLAM, as well as provably correct refinements of this initial abstract machine to capture relevant implementation details.

References

[1] Berry, G., and G. Boudol, *The Chemical Abstract Machine*, Theoretical Computer Science **96** (1992), 217–248.

[2] Bugliesi, M., G. Castagna, and S. Crafa, "Boxed Ambients", In *Proceedings 4th International Symposium on Theoretical Aspects of Computer Software (TACS'01)*, Lecture Notes in Computer Science **2215** (2001), 38–63, Springer-Verlag.

[3] Cardelli, L., "Wide Area Computation", In *Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*, Lecture Notes in Computer Science **1644** (1999), 10–24, Springer-Verlag.

[4] Cardelli, L., and A. Gordon, "Mobile Ambients", In *Proceedings Foundations of Software Science and Computation Structures (FoSSaCS'98)*, Lecture Notes in Computer Science **1378** (1998), 140–155, Springer-Verlag.

[5] Le Fessant, F., and L. Maranget, "Compiling Join-Patterns", In *Proceedings 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, Electronic Notes in Theoretical Computer Science **16 (3)** (2000), Elsevier.

[6] Fournet, C., "The Join-Calculus: a Calculus for Distributed Mobile Programming", Ph.D. thesis, Ecole Polytechnique, 1998.

[7] Leroy, X., *The ZINC Experiment: an Economical Implementation of the ML Language*, INRIA Technical Report, RR-117, 1990.

[8] Levi, F., and D. Sangiorgi, "Controlling Interference in Ambients", In *Proceedings 27th Annual ACM Symposium on Principles of Programming Languages (POPL'00)* (2000), 352–364, ACM Press.

[9] Levy, J.-J., "Some Results in the Join-Calculus", In *Proceedings 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, Lecture Notes in Computer Science **1281** (1997), Springer-Verlag.

[10] Lopes, L., "On the Design and Implementation of a Virtual Machine for Process Calculi", Ph.D. thesis, University of Porto, 1999.

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

[12] Lopes, L., F. Silva, and V. Vasconcelos, "A Virtual Machine for a Process Calculus", In *Proceedings International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Lecture Notes in Computer Science **1702** (1999), Springer-Verlag.

[13] De Nicola, R., G. Ferrari, and R. Pugliese, *Klaim: a Kernel Language for Agents Interaction and Mobility*, IEEE Transactions on Software Engineering, **24 (5)** (1998), 315–330.

[14] Peskine, G., "JAM : the JoCaml Abstract Machine", Private Communication, 2002.

[15] Sangiorgi, D., and A. Valente, "A Distributed Abstract Machine for Safe Ambients", In *Proceedings 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, Lecture Notes in Computer Science **2076** (2001), 408–420, Springer-Verlag.

[16] Schmitt, A., and J.-B. Stefani, *The M-calculus: a Higher-Order Distributed Process Calculus*, INRIA Technical Report, RR-4361, 2002.

[17] Vitek, J., and G. Castagna, "Seal: a Framework for Secure Mobile Computations", In *Proceedings Workshop on Internet Programming Languages*, Lecture Notes in Computer Science **1686** (1999), Springer-Verlag.

[18] Wojciechowski, P., "Nomadic Pict: Language and Infrastructure Design for Mobile Computation", Ph.D. thesis, University of Cambridge, 2000.

[19] Wojciechowski, P., and P. Sewell, *Nomadic Pict: Language and Infrastructure*, IEEE Concurrency, **8 (2)** (2000), 42–52.

[20] Yoshida, N., and M. Hennessy, "Subtyping and Locality in Distributed Higher-Order Processes", In *Proceedings 10th International Conference on Concurrency Theory (CONCUR'99)*, Lecture Notes in Computer Science **1664** (1999), 557–572, Springer-Verlag.

A Notations and Definitions

Notational Conventions We use the following conventions, in addition to standard notations. Given a syntactic category A, we write A^* (resp. $Queue(A)$) for the set of all sequences (resp. queues) containing elements from A. A typical element of such a queue is written as $a_1 :: \ldots :: a_n$. The empty queue is written as $\bullet$. We also write $A \mapsto B$ to denote the set of finite maps from set A to set B. Given any two location names l and l', predicates $desc(l, l')$ (resp. $ether_child(l, l')$, $sibling(l, l')$) are true iff location l is a strict descendant (resp. ether-sublocation, sibling) of l'. The reflexive and transitive closure of a reduction relation $\longrightarrow$ is written as $\Longrightarrow$.

Syntactic categories The following sorts are necessary for the formal description of the AM state:

$a, b, \ldots$	$\in$ CELL	Cell name
$x, y, z \ldots$	$\in$ VAR	Variable
$r, s, t \ldots$	$\in$ REF	Resource name
$n, \ldots$	$\in$ REF $\cup$ CELL	Resolved name
$\mathsf{a} \ldots$	$\in$ VARCELL $=$ CELL $\cup$ VAR	Variable cell name
$\mathsf{r} \ldots$	$\in$ VARREF $=$ REF $\cup$ VAR	Variable resource name
$d \ldots$	$\in$ DEST $=$ VARCELL $\oplus$ VARCELL	Target name
$d.\mathsf{r}, \ldots$	$\in$ ADRREF $=$ DEST $\times$ VARREF	Addressed resource name
$u, \ldots$	$\in$ NAME $=$ VARCELL $\cup$ VARREF $\cup$ ADRREF	Name
$m, m' \ldots$	$\in$ MNAME	Machine name
$P, Q, R \ldots$	$\in$ PR	Process
$V, W, \ldots$	$\in$ VAL	Value

$E, F, G \ldots$	$\in$ EXPR	Expression
$r\tilde{V} \ldots$	$\in$ MSG = ADRREF $\times$ VAL*	Message
$J, J' \ldots$	$\in$ JOIN = Set(REF $\times$ VAR*)	Join-pattern
$D, D' \ldots$	$\in$ DEF = Set(JOIN $\times$ PR)	Definition
$l_{loc}, \ldots$	$\in$ LOCLOCNM	Local location name
$l = \langle m, l_{loc} \rangle, l', \rrbracket \ldots$	$\in$ LOCNM = MNAME $\times$ LOCLOCNM	Location name
$i_{loc}, \ldots$	$\in$ LOCINTNM	Local internal name
$i = \langle m, i_{loc} \rangle \ldots$	$\in$ INTNM = MNAME $\times$ LOCINTNM	Internal name
$\rho, \rho' \ldots$	$\in$ ENV = NAME $\mapsto$ VAL	Environment
$t, t' \ldots$	$\in$ TIMESTAMP	Timestamp

For an environment ρ, we write $\rho \upharpoonright \mathcal{E}$ the restriction of ρ to set $\mathcal{E}$, and $closure(\rho, P)$ the restriction of ρ to free names occcuring in P. We write $loc_id(\widetilde{V})$ for all location names appearing in $\widetilde{V}$. We also write $rsc_id(\widetilde{V})$ for all resources names defined in $\widetilde{V}$. These notations are typically useful for values $\mathsf{reify}_{\mathsf{membrane}}$ and $\mathsf{reify}_{\mathsf{content}}$.

Location state

α, α'	$\in$ CELLNAME	Internal cell name
$\mathcal{D}, \mathcal{D}'$	$\in$ DEFSPACE = Set(ENV $\times$ DEF)	Local active definitions
$\mathcal{H}, \mathcal{H}'$	$\in$ HEAP = INTNM $\mapsto$ VQUEUE	Message heap
$\mathcal{V}, \mathcal{V}'$	$\in$ VQUEUE = $Queue$(ENV $\times$ VAL*)	Local message queue
$\mathcal{R}, \mathcal{R}'$	$\in$ RQUEUE = $Queue$(ENV $\times$ PR)	Run-queue
$\mathcal{E}, \mathcal{E}'$	$\in$ ETHER = LOCNM	Ether-sublocation
$\mathcal{L}, \mathcal{L}'$	$\in$ SUBLOC = Set(LOCNM)	Membrane-sublocations
$\mathcal{M}, \mathcal{M}'$	$\in$ MSGQUEUE = $Queue$(CMSG*)	Global message queue
$\mathbb{L}$	$\in$ LOCSTATE = CELLNAME $\times$ DEFSPACE $\times$ HEAP $\times$ RQUEUE $\times$ ETHER $\times$ SUBLOC	Location state

We write as $l.\mathcal{D}$ (resp. $l.\mathcal{H}, \ldots, l.\mathcal{L}$) the field $\mathcal{D}$ (resp. $\mathcal{H}, \ldots, \mathcal{L}$) of location l. We also use the notation $l.\mathcal{E} = \sharp$ when the ether-sublocation is not defined, i.e., l is an ether-location. We write $locrec(\mathcal{L})$ for the set of locations in $\mathcal{L}$ and their descendants.

Evaluator

$\mathbb{S}_l$	$\in$ EVAL	Evaluator state
$\llbracket E \rrbracket_{load}$	$\in$ EXPR $\to$ EVAL	Load expression
$\llbracket S_l \rrbracket_{unload}$	$\in$ EVAL $\to$ PR	Unload result

Machine State

N	$\in$ NAMEFACTORY = Set(LOCLOCNM $\cup$ LOCINTNM)	Name factory
O	$\in$ LOOKUPSRV = (INTNM $\mapsto$ Set(LOCNM $\times$ TIMESTAMP)) $\cup$ (LOCNM $\mapsto$ (MNAME $\times$ TIMESTAMP))	Lookup service
L	$\in$ LOCS = LOCNM $\mapsto$ LOCSTATE	Location tree
$\mathbb{M}$	$\in$ MSTATE = MNAME $\times$ NAMEFACTORY $\times$ LOOKUPSRV $\times$ LOCS	Machine state
$msg(d.\mathsf{r}, \widetilde{V}, \rho, O)$	$\in$ CMSG = ADRREF $\times$ VAL* $\times$ ENV $\times$ LOOKUPSRV	Inter-machine communication message

We abbreviate as $l \Subset O(\rho(r))$ the statement $\exists t \in$ TIMESTAMP such that $\langle l,t\rangle \in O(\rho(r))$. The negation is written as $l \not\Subset O(\rho(r))$. We also abbreviate as $O\ \{\rho(r) \mapsto^{+} \langle l,0\rangle\}$ the formula $O\ \{\rho(r) \mapsto O(\rho(r)) \cup \{\langle l,0\rangle\}\}$.

We define functions $best$ and $merge$ as follows:

$$best(\langle i_0,t_0\rangle, \langle i_1,t_1\rangle) \stackrel{def}{=} \langle i_0,t_0\rangle \text{ if } t_0 \geq t_1 \ ;\ \langle i_1,t_1\rangle \text{ if } t_0 \leq t_1$$

$$merge(O_0,O_1)(i), i \in \text{LOCNM} \stackrel{def}{=} best(O_0(i),O_1(i)) \text{ if } i \in \mathsf{dom}\,O_0 \cap \mathsf{dom}\,O_1; \quad O_0(i) \text{ if } i \in \mathsf{dom}\,O_0 \setminus \mathsf{dom}\,O_1 \ ;\ O_1(i) \text{ if } i \in \mathsf{dom}\,O_1 \setminus \mathsf{dom}\,O_0$$

$$merge(O_0,O_1)(i), i \in \text{INTNM} \stackrel{def}{=} \{merge_0(\alpha_0,\alpha_1), \forall(\alpha_0,\alpha_1) \in \Theta^2\} \text{ with } \Theta = O_0(i) \cup O_1(i) \text{ if } i \in \mathsf{dom}\,O_0 \cap \mathsf{dom}\,O_1; \quad O_0(i) \text{ if } i \in \mathsf{dom}\,O_0 \setminus \mathsf{dom}\,O_1 \ ;\ O_1(i) \text{ if } i \in \mathsf{dom}\,O_1 \setminus \mathsf{dom}\,O_0$$

$$merge_0(\alpha_0,\alpha_1) \stackrel{def}{=} best(\alpha_0,\alpha_1) \text{ if fst}\,\alpha_0 = \text{fst}\,\alpha_1 \ ;\ \alpha_0 \text{ otherwise}$$

B AM Reduction rules

Figures B.1 through B.4 summarize the AM reduction rules.

Computation:

M.NIL
$$\frac{}{m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, 0) :: \mathcal{R}\rangle\} \longrightarrow m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\}}$$

M.PRL
$$\frac{}{m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, (P \mid Q))\rangle\} \longrightarrow m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, P) :: \mathcal{R} :: (\rho, Q)\rangle\}}$$

M.IF.THEN
$$\frac{match(\rho(\mu), \rho(V))}{m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, ([\mu = V]\ P\ ,\ Q)) :: \mathcal{R}\rangle\} \longrightarrow m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, P) :: \mathcal{R}\rangle\}}$$

M.IF.ELSE
$$\frac{\neg match(\rho(\mu), \rho(V))}{m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, ([\mu = V]\ P\ ,\ Q)) :: \mathcal{R}\rangle\} \longrightarrow m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, Q) :: \mathcal{R}\rangle\}}$$

M.EVAL
$$\frac{\mathbb{S}_l = [\![E]\!]_{load} \qquad \mathbb{S}_l \Longrightarrow_\beta \mathbb{S}'_l \qquad P = [\![\mathbb{S}'_l]\!]_{unload}}{m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, E) :: \mathcal{R}\rangle\} \longrightarrow m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R} :: (\rho, P)\rangle\}}$$

M.NEW
$$\frac{\rho' = \rho\{n \mapsto i\} \quad i = \langle m, i_{loc}\rangle \quad i_{loc} \notin N}{m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho, (\nu\, n)\ P) :: \mathcal{R}\rangle\} \longrightarrow m : N \oplus \{i_{loc}\} : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, (\rho', P) :: \mathcal{R}\rangle\}}$$

Fig. B.1. The CLAM Reduction Rules (1)

Definitions and Local Communication:

M.DEF
$$\dfrac{\rho' = \rho \upharpoonright \bigcup_{J_i \triangleright P_i \in D} fn(P_i) \setminus rn(J_i) \quad \mathcal{D}' = \mathcal{D} \oplus (\rho', D) \qquad \mathcal{H}' = \mathcal{H}\,\{\rho(r) \mapsto \bullet\}_{r \in dn(D)} \quad O' = O\,\{\rho(r) \mapsto^{+} \langle l, 0\rangle\}_{r \in dn(D)}}{m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \langle \alpha, (\rho, \langle D\rangle) :: \mathcal{R}\rangle\} \longrightarrow m : N : O' : L\{l \underset{-}{\overset{\mathcal{D}':\mathcal{H}'}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\}}$$

M.MSG
$$\left\{\begin{array}{l}\dfrac{\mathcal{H}(\rho(r)) = \mathcal{V} \quad \mathcal{H}' = \mathcal{H}\,\{\rho(r) \mapsto \mathcal{V} :: (\rho, \widetilde{V})\}}{m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \left\langle \alpha, (\rho, r\widetilde{V}) :: \mathcal{R}\right\rangle\} \longrightarrow m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}'}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\}} \\[2em] \dfrac{\rho(r) \notin \mathsf{dom}\,\mathcal{H}}{m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \left\langle \alpha, (\rho, r\widetilde{V}) :: \mathcal{R}\right\rangle\} \longrightarrow m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V})\right\rangle\}}\end{array}\right.$$

M.COMM
$$\dfrac{\begin{array}{c}(\rho, D) \in \mathcal{D} \quad D = \ldots ; J \triangleright P ; \ldots \quad J = r_1\widetilde{x_1} \mid \ldots \mid r_q\widetilde{x_q} \\ \left[\mathcal{H}(\rho(r_i)) = (\rho_i, \widetilde{V_i}) :: \mathcal{V}_i\right]_{1\le i\le q} \quad \rho' = \left[\rho \cup \bigcup_{1\le i\le q} \rho_i\right]\{\widetilde{x_i} \mapsto \widetilde{V_i}\}_{1\le i\le q} \quad \mathcal{H}' = \left[\mathcal{H}\,\{\rho(r_i) \mapsto \mathcal{V}_i\}\right]_{1\le i\le q}\end{array}}{m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \longrightarrow m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}'}{\longmapsto}} \langle \alpha, \mathcal{R} :: (\rho', P)\rangle\}}$$

Cell Creation:

M.CELL
$$\dfrac{\begin{array}{c}l' = \langle m, l'_{loc}\rangle \quad l'' = \langle m, l''_{loc}\rangle \quad l'_{loc} \notin N \quad l''_{loc} \notin N \\ \rho' = closure(\rho, P) \quad \rho'' = closure(\rho, Q) \quad \mathcal{L}'_0 = \mathcal{L}_0 \cup \{l'\}\end{array}}{\begin{array}{c}m : N : O : L\{l_0 \underset{l:\mathcal{L}_0}{\overset{-}{\longmapsto}} \langle \alpha_0, \mathcal{R}_0\rangle, l \underset{-}{\overset{-}{\longmapsto}} \langle \sharp, (\rho, b(P)[Q]) :: \mathcal{R}\rangle\} \longrightarrow \\ m : N \oplus \{l'_{loc}, l''_{loc}\} : O \oplus \{l' \mapsto \langle m, 0\rangle, l'' \mapsto \langle m, 0\rangle\} : \\ L\{l_0 \underset{l:\mathcal{L}'_0}{\overset{-}{\longmapsto}} \langle \alpha_0, \mathcal{R}_0\rangle, l \underset{-}{\overset{-}{\longmapsto}} \langle \sharp, \mathcal{R}\rangle, l' \underset{l'':\varnothing}{\overset{\varnothing:\varnothing}{\longmapsto}} \langle \rho(b), (\rho', P)\rangle, l'' \underset{\sharp:\varnothing}{\overset{\varnothing:\varnothing}{\longmapsto}} \langle \sharp, (\rho'', Q)\rangle\}\end{array}}$$

M.CELL$_\top$
$$\dfrac{l = \langle m, l_{loc}\rangle \quad l_{loc} \notin N \quad \rho' = closure(\rho, P) \quad \rho'' = closure(\rho, Q)}{\begin{array}{c}m : N : O : L\{l_\top \underset{\sharp:\varnothing}{\overset{-}{\longmapsto}} \langle \sharp, (\rho, b(P)[Q])\rangle\} \longrightarrow \\ m : N \oplus \{l_{loc}\} : O \oplus \{l \mapsto \langle m, 0\rangle\} : L\{l_\top \underset{l:\varnothing}{\overset{-}{\longmapsto}} \langle \rho(b), (\rho', P)\rangle, l \underset{\sharp:\varnothing}{\overset{\varnothing:\varnothing}{\longmapsto}} \langle \sharp, (\rho'', Q)\rangle\}\end{array}}$$

Cell Mobility:

M.PASSIV
$$\dfrac{\begin{array}{c}\rho(a) = \alpha \quad l \in \mathcal{L}_0 \quad \mathcal{L}'_0 = \mathcal{L}_0 \backslash \{l\} \\ \left\langle \mathsf{reify_{membrane}}(\mathcal{D}, \mathcal{H}, \mathcal{R}), \mathsf{reify_{content}}(l_e, \mathcal{L}), L'\right\rangle = extract(l, \mathcal{D}, \mathcal{H}, \mathcal{R}, l_e, \mathcal{L}, L)\end{array}}{\begin{array}{c}m : N : O : L\{l \underset{l_e:\mathcal{L}}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \langle \alpha, (\rho, \mathsf{pass}_a\ V) :: \mathcal{R}\rangle, l_0 \underset{l':\mathcal{L}_0}{\overset{-}{\longmapsto}} \langle \alpha_0, \mathcal{R}_0\rangle, l' \underset{-}{\overset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \longrightarrow \\ m : N : O : L'\{l_0 \underset{l':\mathcal{L}'_0}{\overset{-}{\longmapsto}} \langle \alpha_0, \mathcal{R}_0\rangle, l' \underset{-}{\overset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, V\,\mathsf{reify_{membrane}}(\mathcal{D}, \mathcal{H}, \mathcal{R})\ \mathsf{reify_{content}}(l_e, \mathcal{L}))\right\rangle\}\end{array}}$$

M.MEMBRANE
$$\dfrac{\begin{array}{c}\left\langle \mathcal{D}', \mathcal{H}', \mathcal{R}'\right\rangle = insert_{flat}(\mathsf{reify_{membrane}}(\mathcal{D}_m, \mathcal{H}_m, \mathcal{R}_m), \mathcal{D}, \mathcal{H}, \mathcal{R}) \\ O' = O\{\forall D \in \mathcal{D}_m\ \ \forall r \in dn(D) \text{ where } O(\rho(r)) = \{\langle \daleth, t_r\rangle\} \uplus \Theta, \rho(r) \mapsto \{\langle l, t_r\rangle\} \uplus \Theta\}\end{array}}{m : N : O : L\{l \underset{-}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \langle \alpha, (\rho, \mathsf{reify_{membrane}}(\mathcal{D}_m, \mathcal{H}_m, \mathcal{R}_m)\ ()) :: \mathcal{R}\rangle\} \longrightarrow m : N : O' : L\{l \underset{-}{\overset{\mathcal{D}':\mathcal{H}'}{\longmapsto}} \langle \alpha, \mathcal{R}'\rangle\}}$$

M.CONTENT
$$\dfrac{\begin{array}{c}\left\langle \mathcal{D}', \mathcal{H}', \mathcal{R}', \mathcal{L}'_0, L'\right\rangle = insert_{sublocs}(\mathsf{reify_{content}}(l_e, \mathcal{L}), \mathcal{D}, \mathcal{H}, \mathcal{R}, \mathcal{L}_0, L) \\ O' = O\{\forall D \in l_e.\mathcal{D}\ \ \forall r \in dn(D) \text{ where } O(\rho(r)) = \{\langle \daleth, t_r\rangle\} \uplus \Theta, \rho(r) \mapsto \{\langle l, t_r\rangle\} \uplus \Theta\} \\ O'' = O'\{\forall l_s \in locrec(\mathcal{L})\ \ \forall D \in l_s.\mathcal{D}\ \ \forall r \in dn(D) \text{ where } O(\rho(r)) = \{\langle \daleth, t_r\rangle\} \uplus \Theta, \rho(r) \mapsto \{\langle l_s, t_r\rangle\} \uplus \Theta\}\end{array}}{\begin{array}{c}m : N : O : L\{l_0 \underset{l:\mathcal{L}_0}{\overset{-}{\longmapsto}} \langle \alpha_0, \mathcal{R}_0\rangle, l \underset{-}{\overset{\mathcal{D}:\mathcal{H}}{\longmapsto}} \langle \sharp, (\rho, \mathsf{reify_{content}}(l_e, \mathcal{L})\ ()) :: \mathcal{R}\rangle\} \longrightarrow \\ m : N : O'' : L'\{l_0 \underset{l:\mathcal{L}'_0}{\overset{-}{\longmapsto}} \langle \alpha_0, \mathcal{R}_0\rangle, l \underset{-}{\overset{\mathcal{D}':\mathcal{H}'}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\}\end{array}}$$

Fig. B.2. The CLAM Reduction Rules (2)

Routing Local Messages:

M.L.MC
$$\frac{l \notin O(\rho(r)) \quad l' \in O(\rho(r))}{\begin{array}{c} m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \left\langle \alpha, (\rho, r\widetilde{V}) :: \mathcal{R} \right\rangle, l' \xmapsto[-]{-} \langle \sharp, \mathcal{R}' \rangle\} \longrightarrow \\ m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, \mathcal{R}' :: (\rho, r\widetilde{V}) \right\rangle\} \end{array}}$$

M.L.CM
$$\frac{l \in O(\rho(r)) \quad l' \notin O(\rho(r))}{\begin{array}{c} m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, (\rho, r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle, l' \xmapsto[-]{-} \langle \sharp, \mathcal{R}' \rangle\} \end{array}}$$

Routing Addressed Messages:

M.A$_1$.MM
$$\frac{\rho(a) = \alpha \quad l \in O(\rho(r))}{m : N : O : L\{l \xmapsto[-]{-} \left\langle \alpha, (\rho, \overline{a}.r\widetilde{V}) :: \mathcal{R} \right\rangle\} \longrightarrow m : N : O : L\{l \xmapsto[-]{-} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle\}}$$

M.A$_1$.EM.L
$$\frac{\rho(a) = \alpha \quad l \in O(\rho(r)) \quad siblings(l, l')}{\begin{array}{c} m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, (\rho, \overline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m : N : O : L\{l \xmapsto[-]{-} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle, l' \xmapsto[-]{-} \langle \sharp, \mathcal{R}' \rangle\} \end{array}}$$

M.A$_1$.EM.R
$$\frac{O_0(\rho(a)) = \langle l_\top, t \rangle \quad l_\top \in O(\rho(r)) \quad O' = merge(O, O_0)}{\begin{array}{c} m : N : O : L\{l_\top \xmapsto[-]{-} \langle \alpha, \mathcal{R} \rangle\} \parallel E\{m' \rightarrow m \mapsto msg(\overline{a}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}\} \longrightarrow \\ m : N : O' : L\{l_\top \xmapsto[-]{-} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle\} \parallel E\{m' \rightarrow m \mapsto \mathcal{M}\} \end{array}}$$

M.A$_1$.CM
$$\frac{\rho(a) = \alpha \quad l \in O(\rho(r))}{\begin{array}{c} m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, (\rho, \overline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle, l' \xmapsto[-]{-} \langle \sharp, \mathcal{R}' \rangle\} \end{array}}$$

M.A$_1$.CC
$$\frac{\rho(a) = \alpha \quad l' \in O(\rho(r))}{\begin{array}{c} m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, (\rho, \underline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, \mathcal{R}' :: (\rho, r\widetilde{V}) \right\rangle\} \end{array}}$$

M.A$_1$.MC
$$\frac{\rho(a) = \alpha \quad l' \in O(\rho(r))}{\begin{array}{c} m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \left\langle \alpha, (\rho, \underline{a}.r\widetilde{V}) :: \mathcal{R} \right\rangle, l' \xmapsto[-]{-} \langle \sharp, \mathcal{R}' \rangle\} \longrightarrow \\ m : N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, \mathcal{R}' :: (\rho, r\widetilde{V}) \right\rangle\} \end{array}}$$

M.A$_1$.EC.L
$$\frac{\rho(a) = \alpha \quad k \in O(\rho(r)) \quad ether_child(k, l) \quad siblings(l, l')}{\begin{array}{c} m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, \mathcal{R} \rangle, l' \xmapsto[-]{-} \left\langle \sharp, (\rho, \underline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m : N : O : L\{l \xmapsto[-]{-} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\ (\underline{a}, r, \widetilde{V})) \right\rangle, l' \xmapsto[-]{-} \langle \sharp, \mathcal{R}' \rangle\} \end{array}}$$

M.A$_1$.EC.R
$$\frac{O_0(\rho(a)) = \langle l_\top, t \rangle \quad k \in O(\rho(r)) \quad ether_child(k, l_\top) \quad O' = merge(O, O_0)}{\begin{array}{c} m : N : O : L\{l_\top \xmapsto[-]{-} \langle \alpha, \mathcal{R} \rangle\} \parallel E\{m' \rightarrow m \mapsto msg(\underline{a}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}\} \longrightarrow \\ m : N : O' : L\{l_\top \xmapsto[-]{-} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\ (\underline{a}, r, \widetilde{V})) \right\rangle\} \parallel E\{m' \rightarrow m \mapsto \mathcal{M}\} \end{array}}$$

Fig. B.3. The CLAM Reduction Rules (3)

Routing Addressed Messages (con't):

M.A2.IN.L
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad desc(k,l) \quad siblings(l,l')}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \underline{\overline{b}}.r\widetilde{V}) :: \mathcal{R}'\right\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\,(\underline{\overline{b}}, r, \widetilde{V}))\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \end{array}}$$

M.A2.IN.R
$$\frac{O_0(\rho(b)) = \langle k,t\rangle \quad desc(k,l_\top) \quad O' = merge(O,O_0)}{\begin{array}{c} m : N : O : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m' \rightarrow m \mapsto msg(\underline{\overline{b}}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}\} \longrightarrow \\ m : N : O' : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\,(\underline{\overline{b}}, r, \widetilde{V}))\right\rangle\} \parallel E\{m' \rightarrow m \mapsto \mathcal{M}\} \end{array}}$$

M.A2.MC
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad desc(k,l)}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \left\langle \alpha, (\rho, \underline{\overline{b}}.r\widetilde{V}) :: \mathcal{R}\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, \underline{\overline{b}}.r\widetilde{V})\right\rangle\} \end{array}}$$

M.A2.ME.L
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad k \neq l \quad \neg\, desc(k,l) \quad siblings(l,l')}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, (\rho, \underline{\overline{b}}.r\widetilde{V}) :: \mathcal{R}\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, \underline{\overline{b}}.r\widetilde{V})\right\rangle\} \end{array}}$$

M.A2.ME.R
$$\frac{\begin{array}{c} O(\rho(b)) = \langle l'',t''\rangle \quad O(l'') = \langle m',t'\rangle \quad m' \neq m \\ O' = O\{\daleth \mapsto \langle m',0\rangle\} \quad \daleth = \langle m, \daleth_{loc}\rangle \quad \daleth_{loc} \notin N \\ O'' = O'\{\forall l \in loc_id(\widetilde{V}) \text{ where } O'(l) = \langle m, t_l\rangle\ , \\ (\forall r \in rsc_id(\widetilde{V}) \text{ where } O'(\rho(r)) = \{\langle l, t_{lr}\rangle\} \uplus \Theta_{lr}\ ,\ \rho(r) \mapsto \{\langle \daleth, t_{lr}+1\rangle\} \uplus \Theta_{lr}), \\ l \mapsto \langle m', t_l + 1\rangle\} \end{array}}{\begin{array}{c} m : N : O : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, (\rho, \underline{\overline{b}}.r\widetilde{V}) :: \mathcal{R}\right\rangle\} \parallel E\{m \rightarrow m' \mapsto \mathcal{M}\} \longrightarrow \\ m : N \oplus \{\daleth_{loc}\} : O'' : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m \rightarrow m' \mapsto \mathcal{M} :: msg(\underline{\overline{b}}.r, \widetilde{V}, \rho, O'')\} \end{array}}$$

M.A2.OUT
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad k \neq l \quad \neg\, desc(k,l)}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \underline{\overline{b}}.r\widetilde{V}) :: \mathcal{R}'\right\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{o}\,(\underline{\overline{b}}, r, \widetilde{V}))\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \end{array}}$$

M.FORWARD
$$\frac{O_0(\rho(b)) = \langle k,t\rangle \quad O(k) = \langle m',t'\rangle \quad m \neq m' \quad O' = merge(O,O_0)}{\begin{array}{c} m : N : O : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m_0 \rightarrow m \mapsto msg(\underline{\overline{b}}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}_0,\ m \rightarrow m' \mapsto \mathcal{M}\} \longrightarrow \\ m : N : O' : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m_0 \rightarrow m \mapsto \mathcal{M}_0,\ m \rightarrow m' \mapsto \mathcal{M} :: msg(\underline{\overline{b}}.r, \widetilde{V}, \rho, O')\} \end{array}}$$

Fig. B.4. The CLAM Reduction Rules (4)