An Abstract Machine
for a Higher-Order Distributed Process Calculus

Florence Germain, Marc Lacoste, and Jean-Bernard Stefani

1 France Telecom R & D, 28, Chemin du Vieux Chêne, 38243 Meylan Cedex, France
{florence.germain, marc.lacoste}@rd.francetelecom.com
2 INRIA Rhône-Alpes, 655, Avenue de l’Europe, 38334 Saint-Ismier Cedex, France
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, 8], the Seal calculus [16], Nomadic Pict [18], the Dr λ calculus [19], DiTyCo [10], Klaim [12], and the different variants of Mobile Ambients [4] such as Safe Ambients [7], 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 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. This is precisely the aim of the M-calculus 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 α(P)[Q] for the cell named α 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 asynchronous communication, and dynamic binding, make the \(M\)-calculus an interesting candidate for a realistic distributed and mobile programming model. However, one of the prime requirements of such a model is to 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. The implementation is also succinctly and accurately described. One of the by-products of this work is to show that, even though the \(M\)-calculus constitutes a non-trivial, higher-order extension of the (distributed) Join calculus, a distributed abstract machine for the \(M\)-calculus is in fact no more complex than an equivalent machine for the Join calculus.

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 The \(M\)-calculus

Syntax The syntax of the \(M\)-calculus, given in Figure 1, is a mix of call-by-value \(\lambda\)-calculus and of Join calculus. From the former, we have \(\lambda\)-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 \(\pi\)-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 \(r\tilde{V}\), where \(r\) is a resource name (the message target) and \(\tilde{V}\) 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 \(d.r\tilde{V}\), where \(d\) denotes the target cell and \(r\) denotes a resource, can be used for communication between different cells. If \(d\) is of the form \(\sigma\), (resp. \(a\)) 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 \(e\)), 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 \(\emptyset\), a name \(u\) (which can be a cell name, a resource name, or an addressed resource name), a tuple of values \(\tilde{V}\), or a \(\lambda\)-abstraction. Note that it is not possible to abstract over resource names appearing in definitions: \(x. (x \triangleright P)\) is not a licit \(M\)-calculus term.

The \(pass\) 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 of the entire cell \(a\). The argument of the \(pass\)
Syntax:

- $S :=$ [system] $V ::= [value]
- $e[P]$ [root cell] $(\, )$ [null value]
- $\forall n. S$ [restriction] $u$ [name]
- $P ::= [process]$ $d$ [target]
- $0$ [null process] $\lambda x.P$ [abstraction]
- $V$ [value] $D ::= [definition]
- $a(P)[P]$ [cell] $\top$ [empty definition]
- $(P \mid P)$ [parallel composition] $J \vdash P$ [reaction rule]
- $(P P)$ [application] $D; D$ [composition]
- $\forall n. P$ [restriction]
- $(\mu = V)[P, P]$ [conditional]
- $(D)$ [definition] $J ::= [join pattern]
- \text{pass}_n V$ [passivation] $r\vec{x}$ [message pattern]
- $| J \mid J$ [synchronization]

Names:

- $u ::= [name]
- a ::= [cell name]
- r ::= [variable resource name]
- $\forall n. S$ [resolution name]
- $a ::= [cell name]$
- $r ::= [variable resource name]$
- $n ::= [resolved name]$
- $d ::= [target name]$
- $r ::= [variable resource name]$
- $\forall n. S$ [located resource]
- $| x ::= [variable]$
- $\forall n. S$ [located resource]
- $\forall n. S$ [located resource]
- $\forall n. S$ [located resource]
- $\forall n. S$ [located resource]

Free names:

- $fn(a) = fn(\pi) = fn(\pi) = \{a\}$
- $fn(r) = \{r\}$
- $fn(d r) = fn(d) \cup \{r\}$
- $fn(\lambda r) = \emptyset$
- $fn(\bot) = fn(\lambda x.P) = fn(P) \setminus \{x\}$
- $fn(n P) = fn(P) \setminus \{n\}$
- $fn(P Q) = fn(P) \cup fn(Q)$
- $fn(D D') = fn(D) \cup fn(D')$
- $fn(\top) = \emptyset$
- $fn(\top) = fn(P \setminus fn(J) \cup dn(J))$
- $fn(r_1 x_1 \ldots r_n x_n) = fn(r_1 x_1) \cup \ldots \cup fn(r_n x_n)$
- $fn(r(x_1, \ldots, x_r)) = \{r, x_1, \ldots, x_r\}$
- $fn(a P)[Q] = \{a\} \cup fn(P) \cup fn(Q)$
- $fn(P Q) = fn(P) \cup fn(Q)$
- $fn(\mu = V)[P, Q] = fn(\mu) \cup fn(P) \cup fn(Q) \cup fn(V)$
- $fn(\text{pass}_n V) = \{a\} \cup fn(V)$
- $fn(n S) = fn(S) \setminus \{n\}$
- $fn(\text{pass}_n V) = fn(P)$
- $fn(V_1, \ldots, V_r) = fn(V_1) \cup \ldots \cup fn(V_r)$

Defined local names:

- $dln(n P) = dln(P) \setminus \{n\}$
- $dln(PQ) = \emptyset$
- $dln(D) = dln(D)$
- $dln(D D') = dln(D) \cup dln(D')$
- $dln(T) = \emptyset$
- $dln(J \vdash P) = dln(J)$
- $dln(a P)[Q] = \emptyset$
- $dln(P Q) = dln(P) \cup dln(Q)$
- $dln(\mu = V)[P, Q] = \emptyset$
- $dln(\text{pass}_n V) = \emptyset$
- $dln(\text{pass}_n V) = dln(S) \setminus \{n\}$
- $dln(V) = \emptyset$
- $dln(\text{pass}_n V) = \emptyset$

Active cells:

- $\text{cells}(n P) = \text{cells}(P) \setminus \{n\}$
- $\text{cells}(P Q) = \emptyset$
- $\text{cells}(D) = \emptyset$
- $\text{cells}(a P)[Q] = \{a\} \cup \text{cells}(P) \cup \text{cells}(Q)$
- $\text{cells}(P Q) = \text{cells}(P) \cup \text{cells}(Q)$
- $\text{cells}(\mu = V)[P, Q] = \emptyset$
- $\text{cells}(\text{pass}_n V) = \emptyset$
- $\text{cells}(\text{pass}_n V) = \text{cells}(S) \setminus \{n\}$
- $\text{cells}(\text{pass}_n V) = \text{cells}(P)$
- $\text{cells}(V) = \emptyset$
- $\text{cells}(O) = \emptyset$
- $\text{cells}(x(P)[Q]) = \emptyset$

Fig. 1. The M-calculus: Syntax, Names, and Active Cells
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 \( \mathbb{M} \)-calculus as illustrated by the examples found at the end of the section.

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

\[
\mathcal{E} ::= \cdot \mid \mathcal{E} \mathcal{V} \mid \mathcal{P} \mathcal{E} \mid \nu \mathcal{V} \mathcal{E} \mid (\mathcal{E} \mid \mathcal{P}) \mid \alpha(\mathcal{P}[\mathcal{E}]) \mid \alpha(\mathcal{E}[\mathcal{P}]) \mid \mathcal{V}[\mathcal{E}]
\]

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, \( dn(r_1 \bar{x}_1 | \ldots | r_q \bar{x}_q) = \{ r_1, \ldots, r_q \} \). 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 1. 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 \( \theta \). We use \( \bar{t} \) to denote finite tuples of terms \( (t_1, \ldots, t_q) \). When tuples \( \bar{t} \) and \( \bar{u} \) are of the same size, we note \( \{ \bar{t}/\bar{u} \} \) the substitution \( \{ t_1/u_1, \ldots, t_q/u_q \} \). 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 \( \alpha \)-conversion is noted \( P =_\alpha Q \). We recall that in \( \nu \mathcal{V}. P \), \( \lambda x. P \), and \( r_1 \bar{x}_1 | \ldots | r_q \bar{x}_q \cdot P \), the names and variables \( n, x \), and \( x_{ij} \) are bound in \( P \).

Operational Semantics

The operational semantics of the \( \mathbb{M} \)-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 \( \text{S.NU}. \text{PAR} \) to \( \text{S.CONTEXT} \) given in Figure 2, where the parallel composition operator \( \| \) for processes, and the composition operator \( ; \) for definitions are taken to be commutative and associative, with \( \emptyset \) and \( \top \) as their neutral elements, respectively. The structural rules comprise scope extrusion rules for the restriction operator and standard rules for equivalence under \( \alpha \)-conversion and congruence for evaluation contexts.

The reduction relation for the \( \mathbb{M} \)-calculus, \( \rightarrow \), is defined as the smallest relation that satisfies the rules given in Figure 2. The computing rules for the conditional branch make use of the \( \text{match}() \) predicate defined by:

\[
\text{match}([], V) \quad \text{match}(u, u) \quad \text{match}(\alpha, a. \bar{r}) \quad \text{match}(\bar{r}. a. \bar{r}) \quad \text{match}(\bar{r}. a. \bar{r})
\]

Rule \( \text{R.RES} \) is similar to the \( \text{JOIN} \) 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 \( \text{R.PASSIV} \) is novel and defines the semantics of the \( \text{PASSIV} \) construct. An instruction \( \text{PASSIV}_a V \) 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 \( \bar{t} \) for either \( \bar{t} \) or \( \bar{u} \). 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 \( \text{R.A.MF} \), \( \text{CMF} \), \( \text{EMF} \), \( \text{CF} \), \( \text{MCF} \)). 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 \( \bar{t} \) port (rules \( \text{R.A.ECF} \) and \( \text{R.A.IN} \)). Messages within a cell membrane can flow freely inside the cell content or outside the cell (rules \( \text{R.A.MTC} \), \( \text{MTE} \)). 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 \( \bar{t} \) port (rule \( \text{R.A.CTE} \)). The routing rules in the \( \mathbb{M} \)-calculus correspond to elementary routing steps, that enforce the two following principles: all messages are routed one boundary at a time (external 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.
Structural Equivalence Rules:

\[ \begin{align*}
S.NU.PAR & \quad \frac{n \notin \text{fn}(Q)}{(\nu m.P)\mid Q \equiv \nu m.P\mid Q} \\
S.NU.MEMB & \quad \frac{n \notin \text{fn}(Q) \land n \neq a}{a(\nu m.P)\mid Q \equiv \nu m.a(P)\mid Q} \\
S.NU.CONT & \quad \frac{n \notin \text{fn}(Q) \land n \neq a}{a(P)\mid \nu m.Q \equiv \nu m.a(P)\mid Q} \\
S.NU.TOP & \quad \frac{e\mid \nu m.P \equiv \nu m.e\mid P}{}
\end{align*} \]

Reduction Rules: Computing

\[ \begin{align*}
R.BETA & \quad \frac{(\lambda x.P)V \rightarrow P\{^V x\}}{}
\end{align*} \]

\[ \begin{align*}
R.IF.THEN & \quad \frac{\text{match}(\mu, V)}{([\mu = V]P,Q) \rightarrow P} \\
R.IF.ELSE & \quad \frac{\text{match}(\mu, V)}{([\mu = V]P,Q) \rightarrow Q} \\
R.PASSIV & \quad \frac{a(\text{pass}_a V \mid P)\mid Q \rightarrow V(\lambda P)[\lambda Q]}{}
\end{align*} \]

Reduction Rules: Routing Local Messages

\[ \begin{align*}
R.L.MC & \quad \frac{r \notin \text{dlm}(P)}{a(P)\mid rV\mid Q \equiv a(P)\mid Q \mid rV} \\
R.L.CM & \quad \frac{r \notin \text{dlm}(P)}{a(P)\mid rV\mid Q \equiv a(P)\mid Q \mid rV}
\end{align*} \]

Reduction Rules: Routing Addressed Messages

\[ \begin{align*}
R.A.MF & \quad \frac{r \in \text{dlm}(P)}{a(P)\mid \overline{r}V\mid Q \equiv a(P)\mid Q \mid \overline{r}V} \\
R.A.EMF & \quad \frac{r \in \text{dlm}(P)}{\overline{r}V\mid a(P)\mid Q \equiv a(P)\mid Q \mid \overline{r}V} \\
R.A.CMF & \quad \frac{r \in \text{dlm}(P)}{a(P)\mid Q \mid \overline{r}V \rightarrow a(P)\mid Q \mid rV} \\
R.A.CF & \quad \frac{r \in \text{dlm}(P)}{a(P)\mid Q \mid a\overline{r}V \rightarrow a(P)\mid Q \mid rV} \\
R.A.CMF & \quad \frac{r \in \text{dlm}(Q)}{a(P)\mid Q \mid \overline{r}V \rightarrow a(P)\mid Q \mid rV} \\
R.A.ECF & \quad \frac{r \in \text{dlm}(Q)}{\overline{r}V\mid a(P)\mid Q \equiv a(P)\mid Q \mid \overline{r}V} \\
R.A.MTC & \quad \frac{b \in \text{cells}(P) \cup \text{cells}(Q) \land b \neq a}{a(P)\mid \overline{b}V\mid a(P)\mid Q \equiv a(P)\mid Q \mid \overline{b}V} \\
R.A.MTC & \quad \frac{b \in \text{cells}(Q) \land b \neq a}{a(P)\mid \overline{b}V\mid a(P)\mid Q \equiv a(P)\mid Q \mid \overline{b}V} \\
R.A.CTE & \quad \frac{b \notin \text{cells}(Q) \land b \neq a}{a(P)\mid \overline{b}V\mid a(P)\mid Q \equiv a(P)\mid Q \mid \overline{b}V}
\end{align*} \]

Fig. 2. Structural Equivalence and Reduction Rules
Programming in the $\mathcal{M}$-calculus

The following examples illustrate the main constructs of the calculus.

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

$$Fwd = \{ (d, r, v) \rightarrow d.r.v; \quad o(d, r, v) \rightarrow d.r.v \}$$

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 $\text{go}$ construct in the Join calculus and the $\text{in}$ and $\text{out}$ capabilities in Mobile Ambients). Cell $Q^m(a)$ below can be moved to a different cell:

$$Q^m(a) = a(Fwd | (\text{go } u \triangleright \text{Go}(a, u)))[Q]$$

$$\text{Go}(a, u) = \text{pass}_u \lambda p.q.(\pi_\text{enter} \lambda a(p())[q()])$$

A $(\text{go } u)$ 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 $\text{Enter}(u)$ below to allow the insertion of a new cell in its content:

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

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

$$Q^*(a) = \nu s.o.n.a(Fwd | \{ \text{suspend} | on \triangleright S(a, s); \text{resume} | s f \triangleright R(a, f, on); \text{open} \triangleright O(a); \text{update} f \triangleright U(a, f) \} )$$

$$S(a, s) = \text{pass}_s \lambda p.q.a(p()[s][q()][0])$$

$$R(a, f, on) = \text{pass}_a \lambda p.q.a(p())[on][f()[q()]]$$

$$O(a) = \text{pass}_a \lambda p.q.q()$$

$$U(a, f) = \text{pass}_a \lambda p.q.a(f()[q()])$$

3 The $\mathcal{M}$-calculus Abstract Machine

We now describe a distributed abstract machine (AM) for the $\mathcal{M}$-calculus called CLAM (CeLular Abstract Machine). We make three slight modifications to the source language. (1) To allow a clear identification of a CLAM component specialized in the evaluation of functional terms, we introduce the syntactic category of expressions, which corresponds to processes and values in the $\mathcal{M}$-calculus grammar. (2) We add two special kinds of values, $\text{refy}_{\text{content}}$ and $\text{refy}_{\text{context}}$, respectively devoted to the refication of the membrane and of the content of a cell. (3) In the $\mathcal{M}$-calculus, a membrane may comprise a parallel composition of cells. In the sequel, we only consider the 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 [15]. This type system also enforces a property of unicity of cell names, which is crucial for guaranteeing the determinacy of the routing rules in the calculus and in the CLAM. We do not detail this type system here for lack of space, but the description of our abstract machine relies on the “unicity of cell names” assumption.

**Overview** 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 typically corresponds to 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 correspond to processes that constitute the cell membrane. A membrane location has two kinds of child locations: an ether-location (or simply ether), which collects the flat processes found at the top level of the cell content, and subordinate membrane-locations that correspond to subcells 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 $\tau$. For commodity, a virtual root location for the entire system (machines and global ether) named $\tau$ is also introduced (Figure 3).

Each location in the CLAM includes an execution engine and a functional evaluator with a well-defined interface between the two components. The execution engine maintains a local location state. The functional evaluator simply returns values to the execution engine, given functional 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 constructs of the M-calculus, namely, special values $\text{reify}_{\text{control}}$ and $\text{reify}_{\text{content}}$, conditional branches, cells, parallel compositions, restrictions, definitions and passivation instructions.

**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 the reduction rules are described in the appendix.

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 $(\alpha, D, \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{C})$. The location name $l$ is unique at the level of the entire system. The name $\alpha$ of the corresponding cell in the calculus is only defined for membrane-locations. $D$ 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. $R$ denotes the location run-queue, where processes to be executed are stored. Finally, $E$ and $C$ are used to keep track of the sublocations of $I$, respectively of kind ether and membrane (note that $E$ is always a singleton for membrane locations, or empty for ether locations – in which case $E$ is noted $\emptyset$).

A machine is represented by a tuple $(m : N : O : L)$, 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 lookup service. $L$ keeps track of locations residing on $m$, by mapping location names to $(a, D, H, R, E, C)$ tuples.

**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

$$m : N : O : L \downarrow \{ \frac{D}{l_i} \} \{ \{ \alpha, R \}, \ldots \} \| E \{ m \rightarrow m' \rightarrow M \} \rightarrow$$

$$m : N' : O' : L' \downarrow \{ \frac{D'}{l_i} \} \{ \{ \alpha, R' \}, \ldots \} \| E \{ m \rightarrow m' \rightarrow M :: m_{sg} \} \rightarrow$$

for a machine $m$ where: (i) the state $(\alpha, D, H, R, l_i, C)$ of a location $l$ becomes $(\alpha, D', H', R', l_i, C')$; (ii) a message $m_{sg}$ 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 $\rightarrow$, e.g., $l \rightarrow \{ \alpha, R \}$.

The core of the execution engine mainly deals with sequential computation (see Figure 5). For fairness purposes, a parallel composition $P \parallel Q$ is interpreted by running $P$ immediately, and scheduling $Q$ for later execution (PRL rule). A functional 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 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 unloading to and from the functional evaluator. We do not describe further the functional evaluator. In fact, one could plug in for this purpose any standard $\lambda$ evaluator. 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 $N$ (NEW rule). Note that in rules IF_THEN and IF_ELSE we use an extension of the environment $\rho$ to name patterns and values.

Local communications are specified by rules (see Figure 5) which resemble rules for a Join calculus AM [13], since the involved primitives are direct JOIN-calculus constructs. In DEF, a newly encountered definition is stored in $D$ – 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 $n(J)$ 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 (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 (COM rule).

Operations that change the structure of the forest of locations are cell creation, cell passivation and cell activation. The discussion of the latter is postponed until after the description of the cell migration mechanism. Cell creation (CELL rule) occurs when the process $a(P) | Q$ 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.

Two kinds of routing rules exist: those corresponding to local messages and those corresponding to addressed messages (see Figure 6). Rules for local messages closely follow the corresponding rules in the $M$-calculus. We just give one of the two rules, namely that corresponding to rule $8.1.MC$ in the calculus. Rule $8.1.MC$ allows messages to move down in the location tree: a message $r \langle V \rangle$ can directly enter the ether-sublocation of the current location iff resource $r$ is defined inside that sublocation.

For the routing of addressed messages, the first three rules given in Figure 6 illustrate physically-located routing, while the last two illustrate physically-remote routing (other rules, corresponding to other rules in the $M$-calculus, are similar and are not given in the Figure). In both cases, the messages are routed “as is” until they reach their destination, i.e., the location which implements the target cell. In physically-located
Sequential computation:

\[
\begin{align*}
\text{NIL} & \quad m : N \triangleright L \{ l \mapsto \{ a, (p, 0) :: R \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, R \} \} \\
\text{PRL} & \quad m : N \triangleright L \{ l \mapsto \{ a, (p, (P \mid Q)) \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, (p, P) :: R :: (p, Q) \} \} \\
\text{IF-THEN} & \quad match(p(\mu), p(V)) \\
& \quad m : N \triangleright L \{ l \mapsto \{ a, (p, ([\mu = V \mid P], Q)) :: R \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, (p, P) :: R :: (p, Q) \} \} \\
\text{IF-ELSE} & \quad \neg match(p(\mu), p(V)) \\
& \quad m : N \triangleright L \{ l \mapsto \{ a, (p, ([\mu = V \mid P], Q)) :: R \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, (p, Q) :: R \} \} \\
\text{EVAL} & \quad S_i = [E]_{load} \quad S_i \rightarrow \beta S_i' \quad P = [S_i']_{unload} \\
& \quad m : N \triangleright L \{ l \mapsto \{ a, (p, n) :: R \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, (p, P) :: R \} \} \\
\text{NEW} & \quad \rho' = \rho[n \mapsto i] \quad i = \langle m, i_{loc} \rangle \quad i_{loc} \notin N \\
& \quad m : N \triangleright L \{ l \mapsto \{ a, (p, (\langle m, n \rangle \mapsto P) :: R) \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, (\rho', P) :: R \} \}
\end{align*}
\]

Definitions and local communication:

\[
\begin{align*}
\rho' = \rho \cup \bigcup_{x, P_i \in D} fn(P_i) \setminus m(J_x) \quad D' = D \oplus (\rho', D) \\
\text{DEF} & \quad m : N \triangleright L \{ l \mapsto \{ (p, n) :: R \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ (p, D) :: R \} \} \\
& \quad \text{if } x_i \in dom(H) \\
& \quad \{ \rho(r(x_i)) = \nu \} \rightarrow H = \{ \rho(r) \mapsto \nu :: (p, \bar{V}) \} \\
\text{MSG} & \quad \{ \rho(r(x_i)) \notin dom(H) \} \\
& \quad m : N \triangleright L \{ l \mapsto \{ a, (p, r \bar{V}) :: R \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, (p, r \bar{V}) :: R \} \} \\
& \quad \text{COM} \quad m : N \triangleright L \{ l \mapsto \{ a, R \} \} \longrightarrow m : N \triangleright L \{ l \mapsto \{ a, R \} \}
\end{align*}
\]

Cell creation:

\[
\begin{align*}
\text{CELL} & \quad l' = \langle m, i_{loc} \rangle \quad l'' = \langle m, i''_{loc} \rangle \quad i_{loc} \notin N \quad i''_{loc} \notin N \\
& \quad \rho' = \text{closure}(\rho, P) \quad \rho'' = \text{closure}(\rho, Q) \quad L_0 = L_0 \cup \{ l' \} \\
& \quad m : N \triangleright L \{ \rho \mapsto \{ a, \rho_0 \}, l \mapsto \{ a, (\rho, b(P) \mid Q) :: R \} \} \longrightarrow \\
& \quad m : N \triangleright L \{ \rho \mapsto \{ a, \rho_0 \}, l \mapsto \{ a, (\rho, P) :: R \}, l' \mapsto \{ a, \rho'' \} \} \longrightarrow \\
& \quad m : N \triangleright L \{ \rho \mapsto \{ a, \rho_0 \}, l \mapsto \{ a, (\rho, P), l' \mapsto \{ a, (\rho'', Q) \} \}
\end{align*}
\]

Fig. 5. The CLAM Reduction Rules (1)
rules, we distinguish depending on whether the target location is a sibling of the current location (logically-located rules) or not (logically-remote rules).

- **Logically-located rules**: a message targeted to a location \( l \), and currently located in its surrounding ether, can directly enter its destination, provided the target resource be defined in \( l \) (rule M.A.EMF.L).

- **Logically-remote rules**: message routing between a membrane-location and its surrounding ether is specified by rules M.A.ECF.L and M.A.MTE.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 (rule M.A.MTE.R). When received at the top-location \( l_\tau \) of the target machine, the message is unpacked and added at the end of the run-queue of \( l_\tau \) (M.A.ECF.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 [13].

Cell passivation is described by rule PASS, while cell activation is described by rules CONTROL and CONTENT in Figure 6. The execution of \( \text{pa}_{\text{mem}} 
\) \( V \) induces a double reification of current location \( l \), together with its descendants, using values \( \text{refify}_{\text{control}} \) and \( \text{refify}_{\text{content}} \), 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 \( \text{refify}_{\text{control}} \) and \( \text{refify}_{\text{content}} \) values from the state of the subtree, and to update \( L \). The M.A.MTE.R rule includes a condition specifying the update of the local lookup service before the remote message send operation (note that 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. More precisely, 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 \( \Sigma \) (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.

A final routing rule deals with message forwarding, in the case where a message is sent to a machine which does not contain the target location any more (because of a previous migration). The forwarding rule is defined as:

\[
\text{FORWARD} \quad l_\tau \in T, L \quad O_0(\rho(b)) = \langle k, t \rangle \quad O(k) = \langle m', t' \rangle \quad m \not= m' \quad O' = \text{merge}(O, O_0)
\]

\[
m : N : O : L \{ l_\tau \quad \text{undo} \quad \langle \alpha, \mathcal{R} \rangle \} ||
\]

\[
E\{m_0 \rightarrow m \mapsto m_\Sigma(\Sigma_r, \Sigma, \rho, O_0) \mapsto M_0, m \mapsto m' \mapsto M \mapsto \| m \}
\]

The **CONTROL** rule leads to the unpacking within the current location of the components of the **refify_{control}** process using the \( \text{insert}_{\text{flat}} \) function. 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 **CONTENT** rule leads to the unpacking within the current location of the components of the **refify_{content}**

---

1. Given a location \( l : \{ \alpha, D, H, R, L, \mathcal{L} \} \), we use value \( \text{refify}_{\text{control}}(D, H, R) \) to reify the activable definitions \( D \), message heap \( H \), and run-queue \( R \) of a location, and value \( \text{refify}_{\text{content}}(l, \mathcal{L}) \) to reify all the sublocations of \( l \), i.e., the ether \( L \), the locations of \( \mathcal{L} \), and all their descendants.

2. For instance, this function can be defined by \( \text{insert}_{\text{flat}}(\text{refify}_{\text{control}}(D_m, H_m, R_m), D, H, R) = \langle D \cup D_m, H \cup H_m, R \cup R_m \rangle \).
Routing local messages:

\[
\begin{align*}
\text{M.L.MC} & \\
& \text{If } l' \in O(\rho(\ell)) \quad l \notin O(\rho(\ell)) \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle] \rightarrow \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle]
\end{align*}
\]

Routing addressed messages:

\[
\begin{align*}
\text{M.A.EMF.L} & \\
& \text{siblings}(l, l') \quad l \in O(\rho(k)) \land \forall k \in O(\rho(\ell)) \\
& \text{child}(k, l) \\
& \rho(a) = \alpha \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle] \rightarrow \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle]
\end{align*}
\]

\[
\begin{align*}
\text{M.A.ECF.L} & \\
& \text{siblings}(l, l') \quad O(\rho(b)) = k \\
& O(k) = \langle m, l' \rangle \\
& \text{desc}(k, l') \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle] \rightarrow \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle]
\end{align*}
\]

\[
\begin{align*}
\text{M.A.MTE.L} & \\
& \text{siblings}(l, l') \quad O(\rho(b)) = k \\
& k \neq l \\
& \text{desc}(k, l) \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle] \rightarrow \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, v') :: R' \rangle]
\end{align*}
\]

\[
\begin{align*}
\text{M.A.MTE.R} & \\
& l \in \mathcal{L} \\
& O_0(\rho(b)) = \langle k, t \rangle \\
& O(k) = \langle m, l' \rangle \\
& O' = \text{merge}(O, O_0) \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle] \rightarrow \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, v) :: R \rangle] \rightarrow E\{m \rightarrow m' \rightarrow M\} \\
& \quad \rightarrow \\
& m : N \cup \{2_{\text{occ}}\} : O' : L[l' \xrightarrow{l', y} \langle \alpha, (\rho, v) :: R \rangle] \rightarrow \\
& m : N \cup \{2_{\text{occ}}\} : O' : L[l' \xrightarrow{l', y} \langle \alpha, (\rho, v) :: R \rangle] \rightarrow E\{m' \rightarrow m' \rightarrow M\}
\end{align*}
\]

Cell mobility:

\[
\begin{align*}
\text{PASS} & \\
& \rho(\alpha) = \lambda \in \mathcal{L}_0 \\
& \mathcal{L}'_0 = \mathcal{L}_0 \setminus \{l\} \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, \text{pass}_a, V) :: R \rangle, l_0 \xrightarrow{l'b} \langle \alpha_0, R_0 \rangle, l' \xrightarrow{l'} \langle \gamma, R' \rangle] \rightarrow \\
& m : N : O : L[l_0 \xrightarrow{l'b} \langle \alpha_0, R_0 \rangle, l' \xrightarrow{l'} \langle \gamma, R' \rangle]
\end{align*}
\]

\[
\begin{align*}
\text{CONTROL} & \\
& O' = O \{V \in \mathcal{D}, \forall r \in \text{dr}(D) \text{ where } O(\rho(r)) = \langle \alpha, t \rangle, \rho(r) \mapsto \langle l, t \rangle\} \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, \text{refc}_\mathcal{D}(D, H, R, l, \mathcal{L})) :: R \rangle, \langle \mathcal{D}', \mathcal{H}', R' \rangle \rightarrow \\
& m : N : O : L[l \xrightarrow{l,y} \langle \alpha, (\rho, \text{refc}_\mathcal{D}(D, H, R, l, \mathcal{L})) :: R \rangle, \langle \mathcal{D}', \mathcal{H}', R' \rangle]
\end{align*}
\]

\[
\begin{align*}
\text{CONTENT} & \\
& O'' = O' \{\forall l_0 \in \text{loc}_{\mathcal{C}}(\mathcal{L}) \quad \forall V \in \mathcal{D}, \forall r \in \text{dr}(D) \text{ where } O(\rho(r)) = \langle \alpha, t \rangle, \rho(r) \mapsto \langle l, t \rangle\} \\
& m : N : O : L[l_0 \xrightarrow{l'b} \langle \alpha_0, R_0 \rangle, l' \xrightarrow{l'} \langle \gamma, (\rho, \text{refc}_\mathcal{D}(D, H, R, l, \mathcal{L})) :: R \rangle] \rightarrow \\
& m : N : O' : L[l_0 \xrightarrow{l'b} \langle \alpha_0, R_0 \rangle, l' \xrightarrow{l'} \langle \gamma, R' \rangle]
\end{align*}
\]

Fig. 6. The CLAM Reduction Rules (2)
process using the $\text{insert}_{\text{malloc}}$ 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 CONTROL rule, the lookup service is also updated.

4 The C-VM Implementation

In this section, we briefly describe a centralized implementation of the $\mathcal{M}$-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 $\mathcal{M}$-calculus expressions. Thus, the C-VM platform comes in the form of a compiler which translates $\mathcal{M}$-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, JoCaml or to some extent TyCO [9, 11] 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 [11], reflecting the nesting of process terms in the bytecode 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 [9]: 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 (figure 4). 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 definned 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 between 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, 17], PAN [14], DiTyCO, and work on a Join calculus AM called JAM [13]. Pict provided a starting point by formalizing a concurrent implementation of the \( \pi \)-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 \( \pi \)-calculus channels – on concurrent objects, and definitions respectively – following [4], PAN investigates a migration-oriented approach to express all computations in terms of pure migration. In the C-VM design, DiTyCO provided inspiration for the bytecode structure to directly reflect the organization of the source program. The JAM provided a direct 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 vs. passivation + 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


**A Appendix**

**Notational Conventions** We use the following conventions, in addition to standard notations. Given a syntactic category \( A \), we write \( A^* \) (resp. \( \text{Queue}(A) \)) 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 \( \cdot \). We also write \( A \rightarrow B \) to denote the set of finite maps from set \( A \) to set \( B \). Given any two location names \( l \) and \( l' \), predicates \( \text{desc}(l, l') \) (resp. \( \text{child}(l, l') \), \( \text{sibling}(l, l') \)) are true iff location \( l \) is a strict descendant (resp. child, sibling) of \( l' \). The reflexive and transitive closure of a reduction relation \( \rightarrow \) is written as \( \rightarrow^* \).

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

\[ a, b, \ldots \in \text{CELL} \quad \text{Cell name} \]
\[ x, y, z, \ldots \in \text{VAR} \quad \text{Variable} \]
\[ r, s, t, \ldots \in \text{REF} \quad \text{Resource name} \]
\[ n, \ldots \in \text{REF} \cup \text{CELL} \quad \text{Resolved name} \]
\[ a, \ldots \in \text{VARCELL} = \text{CELL} \cup \text{VAR} \quad \text{Variable cell name} \]
\[ r, \ldots \in \text{VARREF} = \text{REF} \cup \text{VAR} \quad \text{Variable resource name} \]
\[ d, \ldots \in \text{DEST} = \text{VARCELL} \cup \text{VARCELL} \quad \text{Target name} \]
\[ d, r, \ldots \in \text{ADRREF} = \text{DEST} \times \text{VARREF} \quad \text{Addressed resource name} \]
\[ u, \ldots \in \text{NAME} = \text{VARCELL} \cup \text{VARREF} \cup \text{ADRREF} \quad \text{Name} \]
\[ m, m' \ldots \in \text{MNAME} \quad \text{Machine name} \]
\[ P, Q, R \ldots \in \text{PR} \quad \text{Process} \]
\[ V, W, \ldots \in \text{VAL} \quad \text{Value} \]
\[ E, F, G \ldots \in \text{EXPR} \quad \text{Expression} \]
\[ rV, \ldots \in \text{MSG} = \text{ADRREF} \times \text{VAL}^* \quad \text{Message} \]
\[ J, J' \ldots \in \text{JOIN} = \text{Set}(\text{REF} \times \text{VAR}^*) \quad \text{Join-pattern} \]
\[ D, D' \ldots \in \text{DEF} = \text{Set}(\text{JOIN} \times \text{PR}) \quad \text{Definition} \]
\[ l_{loc}, \ldots \in \text{LOCLOCNM} \quad \text{Local location name} \]
\[ l = \{m, l_{loc}\}, \ell, \ldots \in \text{LOCNM} = \text{MNAME} \times \text{LOCLOCNM} \quad \text{Location name} \]
\[ i_{loc}, \ldots \in \text{LOCINTNM} \quad \text{Local internal name} \]
\[ i = \{m, i_{loc}\} \ldots \in \text{INTNM} = \text{MNAME} \times \text{LOCINTNM} \quad \text{Internal name} \]
\[ \rho, \rho' \ldots \in \text{ENV} = \text{NAME} \rightarrow \text{VAL} \quad \text{Environment} \]
\[ t, u \ldots \in \text{TIMESTAMP} \quad \text{Timestamp} \]

For an environment \( \rho \), we write \( \rho[l] \) the restriction of \( \rho \) to set \( \mathcal{E} \), and \( \text{closure}(\rho, P) \) the restriction of \( \rho \) to free names occurring in \( P \). We write \( \text{loc} \_\text{jid}(V) \) for all location names appearing in \( V \). We also write \( \text{res} \_\text{jid}(V) \) for all resources names defined in \( V \). These notations are typically useful for values \( \text{refy}_\text{control} \) and \( \text{refy}_\text{control} \).
Location state

\[ \alpha, \alpha' \in \text{CELLNAME} \quad \text{Internal cell name} \]
\[ \mathcal{D}, \mathcal{D}' \in \text{DEFSpace} = \text{Set}(\text{ENV} \times \text{DEF}) \quad \text{Local active definitions} \]
\[ \mathcal{H}, \mathcal{H}' \in \text{HEAP} = \text{INTNM} \rightarrow \text{VQUEUE} \quad \text{Message heap} \]
\[ V, V' \in \text{VQUEUE} = \text{Queue}(\text{ENV} \times \text{VAL}^*) \quad \text{Local message queue} \]
\[ \mathcal{R}, \mathcal{R}' \in \text{RQUEUE} = \text{Queue}(\text{ENV} \times \text{PR}) \quad \text{Run-queue} \]
\[ \mathcal{E}, \mathcal{E}' \in \text{ETHER} = \text{LOCNM} \quad \text{Ether-sublocation} \]
\[ \mathcal{L}, \mathcal{L}' \in \text{SUBLOC} = \text{Set}(\text{LOCNM}) \quad \text{Membrane-sublocations} \]
\[ \mathcal{M}, \mathcal{M}' \in \text{MSQUEUE} = \text{Queue}(\text{CMST}) \quad \text{Global message queue} \]
\[ L \in \text{LOCSTATE} = \text{CELLNAME} \times \text{DEFSpace} \times \text{LOCATION state} \]
\[ \text{HEAP} \times \text{RQUEUE} \times \text{ETHER} \times \text{SUBLOC} \]

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 \( l. \mathcal{E} = \emptyset \) when the ether-sublocation is not defined, i.e., \( l \) is an ether-location. We write \( \text{locvec}(\mathcal{L}) \) for the set of locations in \( \mathcal{L} \) and their descendants.

Evaluator

\[ S_l \in \text{EVAL} \quad \text{Evaluator state} \]
\[ |E|_{\text{load}} \in \text{EXPR} \rightarrow \text{EVAL} \quad \text{Load expression} \]
\[ |S_l|_{\text{unload}} \in \text{EVAL} \rightarrow \text{PR} \quad \text{Unload result} \]

Machine State

\[ N \in \text{NAMEFACTORY} = \text{Set}(\text{LOCNM} \cup \text{LOCINTNM}) \quad \text{Name factory} \]
\[ O \in \text{LOOKUPSrv} = (\text{INTNM} \rightarrow \text{Set}(\text{LOCNM} \times \text{TIMESTAMP})) \cup (\text{LOCNM} \rightarrow (\text{MNAME} \times \text{TIMESTAMP})) \quad \text{Lookup service} \]
\[ \text{LOC} \in \text{LOCs} = \text{LOCNM} \rightarrow \text{LOCSTATE} \quad \text{Location tree} \]
\[ \mathcal{M} \in \text{MSTATE} = \text{MNAME} \times \text{NAMEFACTORY} \times \text{LOOKUPSrv} \times \text{LOCs} \quad \text{Machine state} \]
\[ \text{msg}(d, r, \mathcal{V}, \rho, O) \in \text{CMST} = \text{ADRRef} \times \text{VAL}^* \times \text{ENV} \times \text{LOOKUPsrv} \quad \text{Inter-machine communication message} \]

We abbreviate as \( l \in O(\rho(r)) \) the statement \( \exists t \in \text{TIMESTAMP} \text{ such that } \langle l, t \rangle \in O(\rho(r)) \). The negation is written as \( l \notin O(\rho(r)) \). We also abbreviate as \( O \{ \rho(r) \} \rightarrow\rangle \{ l, 0 \} \) the formula \( O \{ \rho(r) \} \rightarrow\rangle O(\rho(r)) \cup \{ \langle l, 0 \rangle \} \).

We define functions \( \text{merge} \) and \( \text{best} \) as follows:

\[ \text{merge}(O_0, O_1) \overset{\text{def}}{=} \{ i \in \text{dom}(O_0) \cup \text{dom}(O_1), i \rightarrow \text{best}(O_0(i), O_1(i)) \} \]
\[ \text{best}((t_0, t_0), (t_1, t_1)) \overset{\text{def}}{=} \begin{cases} t_0 \text{ if } t_0 \geq t_1 \\ t_1 \text{ if } t_0 < t_1 \end{cases} \]