A Generic Membrane Model (*)

(Note)

Gérard Boudol

INRIA Sophia Antipolis
BP 93 – 06902 Sophia Antipolis Cedex, France

Abstract

In this note we introduce a generic model for controlling migration in a network of distributed processes. To this end, we equip the membrane of a domain containing processes with some computing power, including in particular some specific primitives to manage the movements of entities from the inside to the outside of a domain, and conversely. We define a π-calculus instance of our model, and illustrate by means of examples its expressive power. We also discuss a possible extension of our migration model to the case of hierarchically organized domains.

1. Introduction

In the past few years, various formal models for explicit distribution and migration of code have been proposed, mostly based on the π-calculus [19] (like π_{1l} [1], Dπ, [14], *lsd*π [20], NomadicPict [23], and the Seal calculus [24]) or on the Mobile Ambients calculus [8] (like the Boxed Ambients [5] and the Safe Mobile Ambients [16]), with the notable exceptions of *Obliq* [6] (which is object-based), Klaim [10] (which is based on the Linda tuple space for communication and coordination), and the Distributed Join-calculus [11]. In all these models, there is a notion of a *domain* – or site, or locality [4, 9], or Ambient – where computations take place. These models also provide, more or less explicitly, primitive constructs for moving code from a domain to another. The moving entities may actually be quite diverse, from domains to closures, including processes (in the π-calculus sense), and even more so if we also consider the various programming languages, often based on Java, that have been proposed for mobile computations (see [12] for a survey).

In most of these models, the migrating entities may move quite freely from one domain to the other, provided that the destination domain is accessible, as in Dπ or the Mobile Ambients calculus for instance. However, it would clearly be better to have some means for a domain to *control* the migration of code through its boundary, for security purposes for instance. Indeed, some of the previously mentioned models offer such means. For instance, in π_{1l} [1], where failures are

(*) Work supported by the MIKADO project of the IST-FET Global Computing Initiative.

taken into account, the success of a migration depends on the status, running or stopped, of the destination domain. In the SEAL calculus [24], moving a seal is achieved by means of higher-order communication, and, as in the π-calculus, the receiver of the moving seal may be in various states, and this allows for controlling migration. Similarly, in the Safe Ambients [16], there is a channel associated with a domain for each capability, and especially **in** and **out**, and this provides a means to reject migration. However, this controlling power remains limited: for instance, one cannot in these models easily program a domain which would forward incoming entities to another destination, or dispatch them to a group of domains.

The M-calculus [22] was specifically designed to provide a notion of a *programmable domain*, in order to equip sites with various semantics, coping for instance with failures, security concerns, resource management, verification of mobile code, and so on. To this end, a domain is enriched with a new component, the controller, which in particular filters incoming and outgoing messages. The M-calculus then introduces a primitive for "passivating", or reifying a domain, which is used to derive various migration behaviours. This **passivate** construct is extremely powerful, and it may seem a little unnatural to have to reify a whole domain in order to accept an incoming agent for instance. In this paper, we shall elaborate on the idea of a programmable domain of the M-calculus, focusing on the migration aspect. In particular, we keep the idea that a domain has, besides its name and content, a controller part, which is the one we are interested in here. We call this part a *membrane*, thus elaborating on the chemical metaphor [2], where a membrane $\{\!|\_|\!\}$ is used as an evaluation context, in which computations can take place. Then the main idea here is to give some "thickness" to membranes, providing them with some computing power, in order to control their *permeability*, that is to control the movements of entities between the outside and the inside of a domain (this is similar to the idea of an "airlock" in the early CHAM [2]).

Our purpose is *not* to design a full programming model, aiming at Turing completeness, but rather to provide a formal model which could be combined with various programming styles regarding the content of domains. This is the sense in which our model aims at being "generic": we shall not specify any particular choice regarding the migrating entities, but we only make a few assumptions about what the content of a domain should support in order to be compatible with our membrane model. As a matter of fact, there will be only two such requirements: the content of a domain should, if migration to the outside is to be performed, be able to send messages to the enclosing membrane. Conversely, if migration to the inside is to be supported, the content of a domain should be able to dynamically accept (as new processes or threads for instance) new entities for execution. Regarding the membrane itself, that is the controller part of a domain, our model relies on a few primitives, mainly for sending messages to other domains, and adding new entities for execution to the content of the domain. Using for instance a π-calculus style for programming the membranes, we show, by means of examples, the expressive power of the proposed model.

2. The Migration Model

As indicated in the Introduction, we shall elaborate on the chemical metaphor (aka multiset rewriting, or structural equivalence), regarding a system of concurrent activities as a chemical solution (a multiset) made of molecules, possibly reacting when they come into contact. In [2] we stated a general "membrane law", asserting that computations can take place in any solution, enclosed into a membrane. In this setting, we may regard a domain as a *named membrane*. However, as we said, we will equip the membrane itself with some computing power, to control its permeability. Then we could use the notation $a\{\, S \,[\, P \,]\!\}$ for a domain named a, with controller S and content P. However, this turns out to be not very readable, and we will rather write $a\{S\}[P]$.

These domains with a programmable membrane are components of concurrent systems called *networks*, that may also contain "packets", that is, messages with a destination. Let us introduce some syntax for networks of domains and packets. Assuming given a set $\mathcal{N}$ of names, ranged over by n, $m \ldots$ and containing the subset $\mathcal{D}$ of domain names, ranged over by a, b, $c \ldots$, the networks are described using the following syntax:

$$A,\ B \ldots \ ::= \ a\{S\}[P] \ \mid \ a\langle M \rangle \ \mid \ (A \parallel B) \ \mid \ (\nu n)A$$

where S is any controller, P is any content, that we call a *process*, $a\langle M \rangle$ is a packet, containing a message M with destination a, $(A \parallel B)$ is the parallel composition, intended to represent the (physical or logical) juxtaposition of domains, and $(\nu n)A$ is, as in the π-calculus, scope restriction, making the name n private to the sub-network A. In this section we shall not consider any syntax for controllers S, processes P, or messages M. We shall rather assume that the controllers are states of a given transition system, and similarly for processes. We distinguish two kinds of transitions: unlabelled ones, $S \to S'$ or $P \to P'$, describe the internal computations a controller or a process may perform, which do not concern its interactions with the environment. The latter will be described by labelled transitions. Regarding the processes, we make the following assumptions:

(1) it should be possible to *dynamically add new processes* for execution to the content of a domain. We represent addition of a new process Q to an existing content P, resulting in a process P', by a labelled transition

$$P \xrightarrow{\downarrow Q} P'$$

(2) it should be possible to *send messages* from the content of a domain to its enclosing membrane. These "upward messages" are denoted $\uparrow M$, and the fact that the process P sends the message M to the enclosing membrane, and becomes P' in doing so, is denoted

$$P \xrightarrow{\uparrow M} P'$$

Typically, using a CCS-like syntax for processes, we would have $P \xrightarrow{\downarrow Q} (P \mid Q)$ and $\uparrow M.P \xrightarrow{\uparrow M} P$, but we shall not further analyze these transitions in the following. Regarding the membranes, controlling the migration from and to the

content of a domain, we shall make similar assumptions:

(3) the membranes have the ability to receive messages (coming either from the inside or the outside of the domain), performing transitions of the form

$$S \xrightarrow{{}^{\downarrow}M} S'$$

(4) the membranes have the ability to perform specific actions to move a process from the membrane to the inside of the domain, and to send messages to another domain in the network. Moving a process from the membrane to the content of a domain is described by transitions of the form

$$S \xrightarrow{\mathbf{in}\langle P\rangle} S'$$

while sending a message in the network corresponds to a transition

$$S \xrightarrow{\mathbf{out}\langle b,M\rangle} S'$$

Let us denote by $\mathcal{S}$, $\mathcal{P}$ and $\mathcal{M}$ respectively the sets of controllers, processes and messages, and let

$$\mathcal{L}_p = \{\, {}^{\downarrow}P \mid P \in \mathcal{P} \,\} \cup \{\, {}^{\uparrow}M \mid M \in \mathcal{M} \,\}$$

and

$$\mathcal{L}_c = \{\, {}^{\downarrow}M \mid M \in \mathcal{M} \,\} \cup \{\, \mathbf{in}\langle P\rangle \mid P \in \mathcal{P} \,\} \cup \{\, \mathbf{out}\langle b, M\rangle \mid b \in \mathcal{D}, M \in \mathcal{M} \,\}$$

be respectively the sets of transition labels for processes and controllers. Then, summarizing, our migration model relies upon a system

$$\mathcal{P}roc \;=\; (\mathcal{P}, \rightarrow, \{\, \xrightarrow{L} \mid L \in \mathcal{L}_p \,\})$$

for processes, and a system

$$\mathcal{C}ontrol \;=\; (\mathcal{S}, \rightarrow, \{\, \xrightarrow{L} \mid L \in \mathcal{L}_c \,\})$$

for controllers. This is the basis on which we define the semantics of domains, and of networks. Notice that we assume here that a message is a passive entity, that does not compute by itself: there is nothing like $M \rightarrow M'$ in our model. As it has become standard (following Milner's presentation of the CHAM [17]), to formulate the operational semantics, we introduce a *structural congruence*. Assuming that a notion of free and bound occurrences of names in controllers, processes and messages is defined, as well as a notion of name substitution $\{n \mapsto m\}$, the structural congruence is the least congruence $\equiv$ on networks satisfying

$$\begin{array}{rcll}
(\nu n)A & \equiv & (\nu m)\{n \mapsto m\}A & m \text{ not free in } A \\
((A \parallel B) \parallel C) & \equiv & (A \parallel (B \parallel C)) & \\
(A \parallel B) & \equiv & (B \parallel A) & \\
((\nu n)A \parallel B) & \equiv & (\nu n)(A \parallel B) & n \text{ not free in } B
\end{array}$$

Then the transition rules are as follows. First, we have to express the fact that computations can occur within a membrane and a domain:

$$\frac{S \rightarrow S'}{a\{S\}[P] \rightarrow a\{S'\}[P]}\ (\mathrm{R1}) \qquad \frac{P \rightarrow P'}{a\{S\}[P] \rightarrow a\{S\}[P']}\ (\mathrm{R2})$$

The next rule (R3) (together with R9) asserts that a packet can be delivered to the membrane of the destination domain, if the membrane is ready to accept it. Similarly, the rule (R4) deals with the case where the message is sent from the content of the domain:

$$\frac{S \xrightarrow{\downarrow M} S'}{(a\langle M\rangle \parallel a\{S\}[P]) \to a\{S'\}[P]} \text{ (R3)} \qquad \frac{S \xrightarrow{\downarrow M} S' \quad P \xrightarrow{\uparrow M} P'}{a\{S\}[P] \to a\{S'\}[P']} \text{ (R4)}$$

The rule (R5) below describes how to add a new process to the content of a domain, by means of the **in** construct, and similarly the rule (R6) describes how to send a message from a membrane to the network, by means of the **out** construct. Observe that we assume that the network is always ready to accept a message from a domain:

$$\frac{S \xrightarrow{\mathbf{in}\langle Q\rangle} S' \quad P \xrightarrow{\downarrow Q} P'}{a\{S\}[P] \to a\{S'\}[P']} \text{ (R5)} \qquad \frac{S \xrightarrow{\mathbf{out}\langle b,M\rangle} S'}{a\{S\}[P] \to (b\langle M\rangle \parallel a\{S'\}[P])} \text{ (R6)}$$

Finally we have:

$$\frac{A \to A'}{(A \parallel B) \to (A' \parallel B)} \text{ (R7)} \qquad \frac{A \to A'}{(\nu n)A \to (\nu n)A'} \text{ (R8)} \qquad \frac{A \equiv B \quad A \to A'}{B \to A'} \text{ (R9)}$$

Notice that a process P is not allowed to compute inside a call $\mathbf{in}\langle P\rangle$. Two important points must be noted:

(1) there is no way for a message to go directly from the outside of a domain to its content, or conversely. Any message to or from a domain has to transit through the membrane, where it is handled (rules R3-R4).

(2) there is no way for an entity to leave the membrane apart from being transmitted as an argument of the **in** and **out** primitives (R5-R6).

This means that migration, that is interactions between domains, by means of messages sent in the network, is always under the control of the membranes. The primitives **in** and **out** are predefined procedures, the semantics of which is described by (R5) and (R6), and which only have a meaning when they are called from within a membrane. Notice also that a membrane has, thanks to the **in** construct, a computing power which is of higher-order with respect to the processes running in a domain. One should also remark that our model adheres to the "*locality principle*", and more precisely to the Dπ [14] "go and communicate" philosophy: there is no reduction rule that involves more than one domain in its left-hand side. In other words, we do not require any synchronization between domains, nor do we assume any "action at a distance". In this way, we do not have to assume that the network has any particular computing power, apart from the one of routing the packets ([1]). For instance, if $S \xrightarrow{\mathbf{out}\langle b,M\rangle} S'$ and $R \xrightarrow{\downarrow M} R'$, the following transition occurs, in two steps, modulo structural congruence:

$$a\{S\}[P] \parallel b\{R\}[Q] \xrightarrow{*} a\{S'\}[Q] \parallel b\{R'\}[P]$$

([1]) and supporting scope extrusion.

We could have adopted this as an atomic transition, thus doing without explicit packets $b\langle M\rangle$. However, considering this sequence as an atomic step would introduce a synchronisation between domains, whereas by our rules a message is allowed to go out of a locality even if its destination domain does not exist. We think it is more realistic to assume that the network is asynchronous – that is, sending a message is never blocked, so that the decision to choose among several messages to send is only local.

Let us further comment on our model, comparing it with the one of Dπ. In Dπ the notion of a packet is implicit: a packet $a\langle M\rangle$ is actually represented as a domain $a[M]$, and the rule for incoming messages is replaced by a structural equivalence:

$$a[P] \parallel a[Q] \equiv a[P \mid Q]$$

We do not adopt such a structural law here, because our purpose is to control incoming processes, and also because this would mean merging the membrane and the content of domains bearing the same name, and this does not fit very well with the idea of a controlled domain. Therefore the "routing" mechanism is non-deterministic, in the case where a message has two possible destinations, like in

$$a\langle M\rangle \parallel a\{S_0\}[Q_0] \parallel a\{S_1\}[Q_1]$$

which seems quite acceptable as a network behaviour. As a consequence of not having confused $a\langle M\rangle$ with $a[M]$ as in Dπ, we lose the capability one has in this calculus to dynamically create localities: in Dπ the $\mathbf{out}\langle a, M\rangle$ construct is written $a :: M$ (or $\mathbf{go}\, a.M$), where M can actually be any process, with the rule

$$b[a :: P \mid Q] \;\rightarrow\; a[P] \parallel b[Q]$$

(Thanks to the previously mentioned structural equivalence, we actually don't even have to mention Q in this rule.) It is certainly useful to have this capability of creating domains, which could be expressed with a new construct $\mathbf{mk\_dom}\langle b, S, P\rangle$, with the rule

$$\frac{S \xrightarrow{\mathbf{mk\_dom}\langle b,T,Q\rangle} S'}{a\{S\}[P] \rightarrow (a\{S'\}[P] \parallel b\{T\}[Q])}$$

However, we shall not use such a construct here. Still keeping the locality principle, a membrane could also be endowed with a capability **kill** of destroying the domain it controls, and one could also imagine some further ways in which membrane may control the content of a domain. However, we shall not explore this here.

3. A π-Calculus Based Model

In order to give more substance to our membrane model, we shall in this section introduce a particular instance, where the membranes are processes written using the π-calculus style. That is, we completely specify a *Control* system here, which esentially amounts to specifying what are the messages and how they are dealt with. This instance of our model will still be "generic", in the sense that we do not further analyze the process part $\mathcal{P}roc$. It should be easy to design a similar, LINDA-based model for instance, replacing the channel based communication by

communication through a tuple space, or an ML-like model, where messages are function calls (however, in this latter case we should have a way of dealing with dynamically incoming function calls), or maybe an object-oriented model, where messages are method invocations.

We assume that the set $\mathcal{N}$ of names contains a set $\mathcal{C}h$, disjoint from $\mathcal{D}$, of *channel names*, ranged over by u, v, $w \ldots$. We also sometimes regard names as *variables*, written x, y, $z \ldots$ We will not make any formal distinction between process variables, channel variables and domain variables, which respectively stand for processes, channel names and domain names (a type system would be useful for that purpose). In the examples we use X, Y, $Z \ldots$ to range over process variables. We also assume that, although they will be used in a similar way as channel names, the constants **in** and **out** of the migration model are *not* names in $\mathcal{N}$. In the π-based instance of our model, the syntax for writing control programs, occurring in the membranes, is the standard syntax of the (asynchronous, polyadic) π-calculus [3, 15, 18], enriched with the specific constructs **in** and **out** of our migration model:

$$
\begin{array}{rcll}
A,\ B \ldots & ::= & a\{S\}[P] \mid a\langle M\rangle \mid (A \parallel B) \mid (\nu n)A & \textit{networks} \\
S,\ T \ldots & ::= & \mathbf{nil} \mid \mathbf{in}\langle P\rangle.S \mid \mathbf{out}\langle a, M\rangle.S \mid M & \textit{controllers} \\
 & \mid & u(\vec{x}).S \mid \mathbf{!}u(\vec{x}).S \mid (S \mid T) \mid (\nu n)S & \\
P,\ Q \ldots & ::= & \cdots & \textit{processes} \\
M & ::= & u\langle \vec{V}\rangle & \textit{messages} \\
V & ::= & n \mid P \mid \cdots & \textit{values}
\end{array}
$$

The messages are tuples of values sent on a channel – which can also be interpreted as the name of a procedure, so that a message is a procedure call, with arguments. The values that can be communicated in messages are either names, or processes, or else values of the usual kind (truth values, integers...), which we freely use, together with the corresponding constructs (like conditional branching), for illustration purposes. Since processes may be arguments in a message, the controller model is of higher-order with respect to the process model. This is crucial for controlling the migration of processes, since this allows us to remove, duplicate, as well as send a process as an argument to various channels for various puposes. As usual, $u(\vec{x}).S$ is a receiver controller, waiting for some message on the channel u. Notice that we restrict replication $\mathbf{!}S$ to receivers. It should also be observed that, since we assumed that **in** and **out** are not names, there is no receiver with these names, according to our assumption that these represent predefined procedures, with a fixed – not programmable – semantics. Regarding the **in** and **out** constructs, we have adopted a "synchronous" style. That is, calling one of these procedures, in $\mathbf{in}\langle P\rangle.S$ and $\mathbf{out}\langle a, M\rangle.S$, involves an explicit continuation S. The reason is that we could not derive, as this can be done for sending names, this synchronization from an asynchronous version, where S is **nil**, because invoking **in** and **out** does not allow sending a return channel.

The operational semantics of this calculus is standard: first, we define the structural equivalence for controllers, still denoted $\equiv$, as the least equivalence

compatible with the static constructs for controllers, that is

$$\frac{S \equiv S'}{(S \mid T) \equiv (S' \mid T)} \qquad \frac{S \equiv S'}{(T \mid S) \equiv (T \mid S')} \qquad \frac{S \equiv S'}{(\nu n)S \equiv (\nu n)S'}$$

and satisfying

$$\begin{array}{rcll} (\nu n)S & \equiv & (\nu m)\{n \mapsto m\}S & m \text{ not free in } S \\ ((S \mid T) \mid U) & \equiv & (S \mid (T \mid U)) & \\ (S \mid T) & \equiv & (T \mid S) & \\ (S \mid \mathbf{nil}) & \equiv & S & \\ ((\nu n)S \mid T) & \equiv & (\nu n)(S \mid T) & n \text{ not free in } T \end{array}$$

Then the transition rules for controllers are the usual communication rules of the π-calculus:

$$(u\langle \vec{V} \rangle \mid u(\vec{x}).S) \to \{\vec{x} \mapsto \vec{V}\}S \qquad (u\langle \vec{V} \rangle \mid \mathbf{!}u(\vec{x}).S) \to (\{\vec{x} \mapsto \vec{V}\}S \mid \mathbf{!}u(\vec{x}).S)$$

together with

$$S \xrightarrow{\downarrow M} (S \mid M) \qquad \mathbf{in}\langle P \rangle.S \xrightarrow{\mathbf{in}\langle P \rangle} S \qquad \mathbf{out}\langle a, M \rangle.S \xrightarrow{\mathbf{out}\langle a, M \rangle} S$$

and

$$\frac{S \to S'}{(S \parallel T) \to (S' \parallel T)} \qquad \frac{S \to S'}{(\nu n)S \to (\nu n)S'} \qquad \frac{S \equiv T \quad S \to S'}{T \to S'}$$

and similarly for the labelled transitions:

$$\frac{S \xrightarrow{L} S'}{(S \parallel T) \xrightarrow{L} (S' \parallel T)} \qquad \frac{S \xrightarrow{L} S' \quad n \text{ not free in } L}{(\nu n)S \xrightarrow{L} (\nu n)S'} \qquad \frac{S \equiv T \quad S \xrightarrow{L} S'}{T \xrightarrow{L} S'}$$

This provides us with the description of the *Control* system we are considering in this section (although in the examples we shall use some further primitives, as we said). Finally the structural congruence over networks has to be extended with

$$\frac{S \equiv T}{a\{S\}[P] \equiv a\{T\}[P]} \qquad \frac{n \neq a \;\&\; n \text{ not free in } P}{a\{(\nu n)S\}[P] \equiv (\nu n)a\{S\}[P]}$$

Notice that we do **not** allow communication at-a-distance, like for instance

$$a\{u\langle \vec{V} \rangle \mid S\}[P] \parallel b\{u(\vec{x}).R \mid T\}[P] \to a\{S\}[P] \parallel b\{\{\vec{x} \mapsto \vec{V}\}R \mid T\}[P]$$

This can only be achieved using $\mathbf{out}\langle b, u\langle \vec{V} \rangle \rangle$ in the a membrane. This means that the channel names, although they are "ubiquitous" and globally known, have a local meaning, provided by the receivers in the membranes.

Now let us see some examples. (We will abbreviate as usual $\mathbf{in}\langle P \rangle.\mathbf{nil}$ and $\mathbf{out}\langle a, M \rangle.\mathbf{nil}$ as $\mathbf{in}\langle P \rangle$ and $\mathbf{out}\langle a, M \rangle$ respectively.) In the π_{1l} [1] and Dπ [14] calculi there is a construction, denoted $\mathbf{go}(b, P)$ ([2]), to send the process P for

([2]) This is written $\mathbf{spawn}(b, P)$ in π_{1l}. In Dπ this is either expressed as a message, denoted $b :: P$, or as an action prefix $\mathbf{go}\, b$.

execution in the remote site b. This can be interpreted as an upward message $^{\uparrow}exit\langle b, P\rangle$ to the enclosing membrane (where *exit* is a name), which is supposed to define a local protocol for sending a process elsewhere. Then, assuming that the procedure to enter a domain is named *enter*, a Dπ domain $a[P]$ can be represented ([3]) as $a\{S\}[P]$ where the membrane contains the following definitions for the ports *enter* and *exit*:

$$\begin{array}{rcl} S & = & \big(\,!enter(X).\mathbf{in}\langle X\rangle \;\mid \\ & & \quad\; !exit(y,X).\mathbf{out}\langle y, enter\langle X\rangle\rangle\,\big) \end{array}$$

For instance, if $P \xrightarrow{\mathbf{go}(b,Q)} P'$, that is $P \xrightarrow{^{\uparrow}exit\langle b,Q\rangle} P'$, and if we let $A = a\{S\}[P]$ and $A' = a\{S\}[P']$, then we have

$$A \to a\{S \mid exit\langle b, Q\rangle\}[P'] \to a\{\mathbf{out}\langle b, enter\langle Q\rangle\rangle \mid S\}[P'] \to b\langle enter\langle Q\rangle\rangle \parallel A'$$

Then, if $R \xrightarrow{^{\downarrow}Q} R'$, and if we let $B = b\{S\}[R]$ and $B' = a\{S\}[R']$, we have

$$A \parallel B \xrightarrow{*} A' \parallel b\{enter\langle Q\rangle \mid S\}[R] \to b\{\mathbf{in}\langle Q\rangle \mid S\}[R] \to A' \parallel B'$$

Therefore this kind of membrane may be called *transparent*, since it does not perform any control on the migrating entities, and just lets them go. Notice that here, as in the following examples, we are implicitly assuming a dynamic binding mechanism: when, for instance, a process Q moving from domains to domains calls the *exit* procedure, it is the local definition, contained in the enclosing membrane, that will be executed, not the one of the site from which the process originates. This is in fact built-in in the π-calculus semantics, where names have a global, or more precisely "ubiquitous" meaning, and in the local communication discipline we have adopted (there are no distant communications, involving two distinct domains for instance).

In π_{1l} [1], a locality may fail, and is able to send messages only if it is in a "running" state. Moreover, a locality offers two public ports *stop* and *ping* respectively to make it fail and to test its status, running or stopped. Then a π_{1l} domain, initially in a "running" status, may be represented as a domain with a membrane of the following kind – assuming that some further computing constructs, like conditional branching, are available:

$$\begin{array}{rcl} S & = & (\nu s)\big(\, s\langle\mathbf{true}\rangle \;\mid \\ & & \quad !enter(X).\mathbf{in}\langle X\rangle \;\mid \\ & & \quad !exit(y,X).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{out}\langle y, enter\langle X\rangle\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid s\langle t\rangle) \;\mid \\ & & \quad !stop.s(t).s\langle\mathbf{false}\rangle \;\mid \\ & & \quad !ping(y,z).s(t).(\mathbf{out}\langle y, z\langle t\rangle\rangle \mid s\langle t\rangle)\,\big) \end{array}$$

The local state $s\langle t\rangle$ of the membrane indicates the status of the domain: running if $t = \mathbf{true}$, and failed otherwise ([4]). Notice that this controller is very much like

([3]) We do not claim that this provides a faithful encoding of the Dπ-calculus. Indeed, the structural equivalence $a[P] \parallel a[Q] \equiv a[P \mid Q]$ of Dπ mentioned above, by which there is only one site bearing a given name in a network, seems difficult to handle in the present setting.

([4]) The message $s\langle t\rangle$ is also used as a lock for the mutual exclusion of the procedures *exit*, *stop* and *ping*.

an object, with a local state and a set of methods. As one can see, nothing can be emitted from a failed domain, since in this case the body of the *exit* procedure is equivalent to **nil**. The syntax for the *ping* construct is slightly different from the one of π_{1l}, because that calculus uses a global communication discipline, while we assume here the local communication discipline of Dπ. Specifically, a $ping\langle b, v\rangle$ message provides as arguments the locality b and the port v at that locality to which to send the result of the invocation of the *ping* procedure, that is the status (**true** for "running", **false** for "failed") of the locality. On the other hand, we have followed the π_{1l} formalization of a failed site, which still accepts incoming processes. We could however have an alternative semantics for the *enter* procedure, which makes the membrane of a failed site *opaque*, namely

$$!enter(X).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{in}\langle X\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid s\langle t\rangle)$$

We would get a different semantics with an "elastic" membrane, on which messages are bouncing when the domain has failed:

$$!enter(X).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{in}\langle X\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{out}\langle a, enter\langle X\rangle\rangle \mid s\langle t\rangle)$$

Indeed, if there are domains with the same name a, the rejected messages get a chance to be accepted somewhere else. As another example, one can imagine that the *exit* procedure, instead of sending directly its argument to its destination as in the "transparent membranes" of Dπ, sends it to a local routing procedure, defined in each node. Then the *exit* procedure would be replaced by

$$\begin{array}{l} !exit(y, X).route\langle y, X\rangle \ \mid \\ !route(y, X).\mathbf{if}\ y = \mathbf{host}\ \mathbf{then}\ \mathbf{in}\langle X\rangle \\ \qquad\qquad \mathbf{else}\ \mathbf{let}\ z = next\_hop\langle y\rangle\ \mathbf{in}\ \mathbf{out}\langle z, route\langle y, X\rangle\rangle \end{array}$$

Here we assume a constant **host**, the (local) value of which is the name of the domain in which it occurs, and we also assume that a local routing table, called *next_hop*, is available, which gives the next node on the route towards the given destination.

One can easily imagine other examples, like forwarding incoming entities to a group of sites, or delegating them for processing to another site, for instance. In this way, one can establish a logical hierarchy of domains, where a site only directly accepts incoming entities if they come from a given group of localities, and otherwise delegates their processing to another site. For instance, assuming that a component value transmitted in a packet is a group g of localities, a site may accept or reject migrating processes depending on a predicate p on groups:

$$!enter(g, X).\mathbf{if}\ p\langle g\rangle\ \mathbf{then}\ \ \mathbf{in}\langle X\rangle\ \mathbf{else}\ \mathbf{nil}$$

Another example of a membrane in which the enter/exit protocol depends on the local state of the membrane is a *counting* membrane. To write it here we assume that the language is enriched with integers. Then, in this example, the local state of the membrane is a counter c which is incremented whenever some entity enters, and decremented when some entity exits the domain. Moreover, there is a bound n on the number of possibly entering entities. The code is, assuming that the membrane becomes opaque when the bound is reached:

$$\begin{array}{rcl} S & = & (\nu c)\big(\, c\langle 0\rangle \ \mid \\ & & \qquad !enter(X).c(z).\mathbf{if}\ z < n\ \mathbf{then}\ (\mathbf{in}\langle X\rangle \mid c\langle z+1\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid c\langle z\rangle) \ \mid \\ & & \qquad !exit(y, X).c(z).(\mathbf{out}\langle y, enter\langle X\rangle\rangle \mid c\langle z-1\rangle)\,\big) \end{array}$$

4. On Migration in a Hierarchically Structured Network

In this section we discuss a possible adaptation of our model to a hierarchical organization of domains. In a hierarchically structured network, a domain may be embedded into another one. Since in our model a domain has two parts – the membrane and the content –, there are a priori two possibilities for the nesting of domains: a domain may be embedded into the membrane, or into the content part of another domain. However, since the membrane only has the rôle of controlling the gates of a domain, only the second possibility looks meaningful. It is not difficult to adapt our model to this case of hierarchically structured networks. However, it is clear that we cannot in this case maintain to the same extent the "parametric" aspect of our model: we have to make strong assumptions about the process language, namely that it allows for the construction of networks of domains as processes. That is, the process syntax should contain the following clauses:

$$P,\ Q \ldots \ ::= \ a\{S\}[P] \ \mid \ a\langle M\rangle \ \mid \ (P \parallel Q) \ \mid \ (\nu n)P \ \mid \ \cdots$$

so that we may represent the dynamic addition of new processes as follows, using parallel composition:

$$Q \xrightarrow{\downarrow P} (Q \parallel P)$$

Then the transition system $\mathcal{P}roc$ for processes should satisfy the rules governing the behaviour of networks, (R1) to (R9). On the other hand, there is nothing to change in the semantics of controllers, and we may use in particular the π-calculus instance of the previous section, with the same examples.

In the rest of this section we assume that the controllers are encoded as π-calculus expressions, as described in the previous section, and we discuss a possible adaptation and extension of the model in this case. With hierarchically structured domains, where domains are processes, we gain the ability of moving domains around, in an "objective" manner (see [8] for this terminology), that is as the content of messages, since a process, and therefore also a domain is a transmissible value. (We actually also have the ability to make a whole network migrate. It is not yet clear how useful this extra expressive power is, but this certainly deserves to be investigated.) This suggests that we could enrich the model, following the ideas of the Mobile Ambients calculus [8] (and of [11, 23]), with the ability for a membrane to send the domain it controls, in a "subjective" manner, as the content of a message. Since the destination of a message is either the enclosing membrane, in the case of upward messages, or a sibling domain, in the case of a packet, the reification of a domain as a message may take two forms ([5]). Moreover, we also have to provide the name of a channel to which the reified domain will be sent. To this end, we add to the model two new primitives $\mathbf{up}\langle u\rangle$ and $\mathbf{to}\langle a, u\rangle$ to the syntax of controllers.

We also notice that, since a process may now be put in parallel with a domain, the entities that may go out of a membrane, by means of the **out** construct, no longer have to be restricted to packets $a\langle M\rangle$. This means that we may now use this construct with a slightly different syntax, namely $\mathbf{out}\langle P\rangle$, so that we

([5]) If we had a π-calculus syntax for processes for instance, there would be another possibility, which is to send a domain on a channel.

now write $\mathbf{out}\langle a\langle M\rangle\rangle$ instead of $\mathbf{out}\langle a, M\rangle$. It is obvious how to generalize the semantics, modifying the rule (R6):

$$\frac{S \xrightarrow{\mathbf{out}\langle Q\rangle} S'}{a\{S\}[P] \to (Q \parallel a\{S'\}[P])} \quad (\mathrm{R6})'$$

This, however, should not be allowed if $a\{S\}[P]$ is a top level domain, component of a network, since a process cannot run at this level. We leave for further research the question of how to cope with this, perhaps using a type system in the style of [7]. Finally the syntax of the π-calculus based model of hierarchically structured domains is therefore as follows:

$A,\ B\ldots$	$::=$	$a\{S\}[P] \mid a\langle M\rangle \mid (A \parallel B) \mid (\nu n)A$	*networks*
$S,\ T\ldots$	$::=$	$\mathbf{nil} \mid \mathbf{in}\langle P\rangle.S \mid \mathbf{out}\langle P\rangle.S$	*controllers*
	$\mid$	$\mathbf{up}\langle u\rangle.S \mid \mathbf{to}\langle a, u\rangle.S$	
	$\mid$	$M \mid u(\vec{x}).S \mid \mathbf{!}u(\vec{x}).S \mid (S \mid T) \mid (\nu n)S$	
$P,\ Q\ldots$	$::=$	$a\{S\}[P] \mid a\langle M\rangle \mid {}^{\uparrow}M \mid (P \parallel Q) \mid (\nu n)P \mid \cdots$	*processes*
M	$::=$	$u\langle\vec{V}\rangle$	*messages*
V	$::=$	$n \mid P \mid \cdots$	*values*

We shall not repeat the semantics of the previously introduced constructs, but only give the meaning of the new primitives:

$$a\{\mathbf{up}\langle u\rangle.S \mid T\}[P] \to {}^{\uparrow}u\langle a\{S \mid T\}[P]\rangle \qquad a\{\mathbf{to}\langle b, u\rangle.S \mid T\}[P] \to b\langle u\langle a\{S \mid T\}[P]\rangle\rangle$$

Again, the first of these transitions should not be allowed if performed by a top level domain, component of a network, since an upward message ${}^{\uparrow}M$ cannot be a component of a network.

This allows us to encode subjective moves, in the style of the Mobile Ambients calculus [8], though not in an atomic way, since our model follows the locality principle. To see this, let us assume that the membranes offer the ports *enter* and *exit*, as in the examples of the previous section. Then the membrane of a domain willing to go out of the enclosing domain should contain a call $\mathbf{up}\langle exit\rangle$. For instance, if

$$S \ = \ (\mathbf{!}exit(X).\mathbf{out}\langle X\rangle \mid S')$$

and

$$T \ = \ (\mathbf{up}\langle exit\rangle.\mathbf{nil} \mid T')$$

then we have

$$\begin{array}{rcl} a\{S\}[b\{T\}[Q] \parallel P] & \to & a\{S\}[{}^{\uparrow}exit\langle b\{T'\}[Q]\rangle \parallel P] \\ & \to & a\{S \mid exit\langle b\{T'\}[Q]\rangle\}[P] \\ & \to & a\{\mathbf{out}\langle b\{T'\}[Q]\rangle \mid S\}[P] \\ & \to & b\{T'\}[Q] \parallel a\{S\}[P] \end{array}$$

Similarly, the membrane of a domain willing to go into a sibling domain called b should contain a call $\mathbf{to}\langle b, enter\rangle$. For instance, if

$$S \ = \ (\mathbf{!}enter(X).\mathbf{in}\langle X\rangle \mid S')$$

and

$$T \;=\; (\mathbf{to}\langle b, \mathit{enter}\rangle.\mathbf{nil} \mid T')$$

then we have

$$\begin{array}{rcl} a\{T\}[Q] \parallel b\{S\}[P] & \rightarrow & b\langle \mathit{enter}\langle a\{T'\}[Q]\rangle\rangle \parallel b\{S\}[P] \\ & \rightarrow & b\{\mathit{enter}\langle a\{T'\}[Q]\rangle \mid S\}[P] \\ & \rightarrow & b\{S \mid \mathbf{in}\langle a\{T'\}[Q]\rangle\}[P] \\ & \rightarrow & b\{S\}[P \parallel a\{T'\}[Q]] \end{array}$$

In this way, we can encode subjective moves à la Mobile Ambients [8]. However, the semantics is not exactly the one of Ambients' movements. For instance, since the transition

$$a\{\mathbf{to}\langle b, u\rangle.S \mid T\}[P] \rightarrow b\langle u\langle a\{S \mid T\}[P]\rangle\rangle$$

can always be performed, a domain willing to go into a sibling one may fail to do so and be stuck in a message $b\langle \mathit{enter}\langle a\{S \mid T\}[P]\rangle\rangle$ if there is actually no sibling domain with name b. As one can see, the granularity of migration is finer in the model we propose than in the Mobile Ambients calculus. Notice also that, unlike in the Mobile Ambients calculus, in our model the critical pairs (or overlapping redexes, or "interferences", following the terminology of [16]) only occur locally. More precisely, the conflicts between capabilities, and especially **up** and **to**, occur in the membrane of a domain. In particular, according to the locality principle, these conflicts do not involve several domains. Then one may hope that this form of non-determinism is simpler to deal with than the one we find in the Mobile Ambients calculus.

5. Conclusion

We have presented a formal model for explicit distribution and migration of code based on the idea of a programmable domain, that was first implemented in the M-calculus [22]. Here we focused on programming the permeability of the membrane, in order to get a simple model, which should be compatible with a wide variety of programming styles. The M-calculus has recently been simplified by Stefani into the calculus of "kells" [21]. It seems that a model like the one proposed here could be encoded into the kell calculus, representing $a\{S\}[P]$ as a pair of nested domains, where the external one, named a, contains an encoding of S, and the internal one, with a private name, contains an encoding of P. Then the kell calculus may be regarded as a low-level model, with respect to the migration calculus we introduced. However, in the kell calculus one has the ability to bypass the discipline enforced in our membrane model. Therefore, we expect that, as usual, one has means in the low-level model to break some desirable properties (that should be expressed for instance as equivalences) that the high-level model is supposed to enforce. In other words, we expect that an encoding of a membrane model, as the one we introduced, into the kell calculus would not be fully abstract. Then we think the membrane model we have presented in this preliminary note deserves to be studied for itself.

Clearly, the membranes of a domain should be given more computing power than the one we considered here. In particular, it is natural to imagine that,

in order to perform some verification on the migrating code, like type checking or security checks (like in the "proof carrying code"), the membrane should be able to deal with a representation ([6]) $`P$ of the code of the incoming processes into some manageable data structure. A first step in this direction is taken by Hennessy et al. in their Safe Dπ-calculus [13], which also explores the idea of a "programmable membrane", from a typing point of view: the "membrane" of a domain in Safe Dπ consists in (higher-order) typed ports, through which incoming code must pass. The type checking which is performed to input incoming code, based on a sophisticated type system, is the way in which migration is controlled in Safe Dπ. It would be interesting to see how this could be expressed in an extended version of our model.

References

[1] R. Amadio, *An asynchronous model of locality, failure, and process mobility,* COORDINATION'97, Lecture Notes in Comput. Sci. 1282 (1997).

[2] G. Berry, G. Boudol, *The chemical abstract machine,* Theoretical Comput. Sci. 96 (1992) 217-248.

[3] G. Boudol, *Asynchrony and the π-calculus,* INRIA Res. Report 1702 (1992).

[4] G. Boudol, I. Castellani, M. Hennessy, A. Kiehn, *Observing localities,* Theoretical Comput. Sci. 114 (1993) 31-61.

[5] M. Bugliesi, G. Castagna, S. Crafa, *Access control for mobile agents: the calculus of Boxed Ambients,* ACM TOPLAS Vol. 26 No. 1 (2004) 57-124.

[6] L. Cardelli, *A language with distributed scope,* Computing Systems Vol. 8, No. 1 (1995) 27-59.

[7] L. Cardelli, G. Ghelli, A. Gordon, *Mobility types for mobile Ambients,* ICALP'99, Lecture Notes in Comput. Sci. 1644 (1999) 230-239.

[8] L. Cardelli, A. Gordon, *Mobile Ambients,* FoSSaCS'98, Lecture Notes in Comput. Sci. 1378 (1998) 140-155.

[9] I. Castellani, *Process Algebras with Localities,* Chapter 15 of the Handbook of Process Algebras (J. Bergstra, A. Ponse and S. Smolka, Eds), Elsevier (2001) 945-1045.

[10] R. De Nicola, G. Ferrari, R. Pugliese, *KLAIM: a kernel langage for agents interaction and mobility,* IEEE Trans. on Software Engineering Vol. 24, No. 5 (1998) 315-330.

([6]) sometimes called "serialization", although a more appropriate terminology would be "gödelization".

[11] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, *A calculus of mobile agents,* CONCUR'96, Lecture Notes in Comput. Sci. 1119 (1996) 406-421.

[12] A. Fuggetta, G.P. Picco, G. Vigna, *Understanding code mobility,* IEEE Trans. on Soft. Eng. Vol. 24 No. 5 (1998) 342-361.

[13] M. Hennessy, J. Rathke, N. Yoshida, *SafeDpi: a language for controlling mobile code,* Comput. Sci. Tech. Rep. 02, University of Sussex (2003).

[14] M. Hennessy, J. Riely, *Resource access control in systems of mobile agents,* Information and Computation 173 (2002) 82-120.

[15] K. Honda, M. Tokoro, *An object calculus for asynchronous communication,* ECOOP'91, Lecture Notes in Comput. Sci. 512 (1991) 133-147.

[16] F. Levi, D. Sangiorgi, *Controlling interference in Ambients,* POPL'00 (2000) 352-364.

[17] R. Milner, *Functions as processes,* Math. Struct. in Comp. Science 2 (1992) 119-141.

[18] R. Milner, *The polyadic π-calculus: a tutorial,* Technical Report ECS-LFCS-91-180, Edinburgh University (1991) Reprinted in Logic and Algebra of Specification, F. Bauer, W. Brauer and H. Schwichtenberg, Eds, Springer Verlag, 1993, 203-246.

[19] R. Milner, J. Parrow, D. Walker, *A calculus of mobile processes,* Information and Computation 100 (1992) 1-77.

[20] A. Ravara, A. Matos, V. Vasconcelos, L. Lopes, *Lexically scoping distribution: what you see is what you get,* Foundations of Global Computing Workshop, ENTCS Vol. 85 (2003).

[21] J.-B. Stefani, *A calculus of kells,* Foundations of Global Computing Workshop, Electronic Notes in Comput. Sci. Vol. 85 (2003).

[22] A. Schmitt, J.-B. Stefani, *The M-calculus: a higher-order distributed process calculus,* POPL'03 (2003) 50-61.

[23] P. Sewell, P. Wojciechowski, *Nomadic Pict: language and infrastructure design for mobile agents,* IEEE Concurrency Vol. 8 No. 2 (2000) 42-52.

[24] J. Vitek, G. Castagna, *Seal: a framework for secure mobile computations,* Workshop on Internet Programming Languages, Lecture Notes in Comput. Sci. 1686 (1999) 47-77.