MIKADO
Mobile Calculi based on Domains

MIKADO Global Computing Project
IST-2001-32222

A Parametric Model of Migration and Mobility Release 1

MIKADO Deliverable D1.2.1

Editor : G. BOUDOL (INRIA)

Author : G. BOUDOL (INRIA)

Classification : Public
Deliverable no. : D1.2.1
Reference : RR/WP1/3
Date : September 2003

Project funded by the European Community under the "Information Society Technologies" Programme (1998–2002)

Elements for a MIKADO Programming Model

Deliverable D1.2.1

Gérard Boudol

September 26, 2003

1 Introduction

This note presents a proposal towards a "migration calculus", along the lines given in the D1.2.0, and following the discussion held during the MIKADO-MYTHS meeting in Brighton, June 5. It does *not* intend to propose a *full* "core programming model". It only deals with the migration features. The main reason is that the intended model should support various instantiations, each relying on a specific basic computing model and a specific programming style (such as the functional, or channel-based style, or tuple-space-based style, as far as the MIKADO project is concerned). As said in D1.2.0, some "salient features" of this hypothetical model should be

- *A migration calculus, based on hierarchically structured domains, and parametric with respect to the communication, coordination and programming constructs.*
- *A notion of programmable membrane, to encompass various semantics of domains.*

Here the intention is to have a simple model with "programmable membranes", as parametric as possible with respect to other computing aspects. In particular, this means that we do not aim at a Turing complete model. As explained in the D1.2.0, there is little justification for the fact that a migration calculus – say, like the Mobile Ambients calculus – should have universal computing power. On the contrary, a migration calculus may borrow some, or even most of its computing power from some well-established computing model.

The "building blocks" of the migration calculus are, not surprisingly, *domains* – also called "localities", "Ambients", etc. –, consisting of a *named membrane* enclosing some computing entities – usually called *processes*, in notation $a[P]$. Part of the meaning of domains is expressed in the fact that a domain is a place where the computation of some process occurs, that is:

$$\frac{P \to P'}{a[P] \to a[P']}$$

The computation occurring inside a domain can, in principle, be of any kind, but this essentially means that there can be many instances of the migration calculus, depending on the computing model used for the content P of a domain. However, as we shall see, there are some requirements on this model. A domain offers some *local resources* – which are the main reasons for moving –, which may be physical, like time and space for computing, or availability of specific devices, or logical. The latter can be for instance a local state, which in turn can have many forms: it can be a shared memory, or, as in a LINDA-like model, a tuple space, or a set of receivers as in CCS or the π-calculus. The logical resources can also comprise libraries of modules (or "packages"), and/or some operating system, some databases, etc. The exact nature of "local resources" one may find in a domain is something that we will not investigate here.

The domains constitute a *network*, in which some entities will migrate from one domain to another. One usually describes a network as a parallel composition of domains, and a preliminary question to examine is: can a domain, and more generally a network, be a computing entity or not? In other words, can domains be

nested, organized in a hierarchic structure, or not? In any case, one certainly has the idea that a network of domains may, in particular, be a model for a real network, made of machines connected by wires or radio waves. In this case, it would be absurd to assume for instance that a running program can be a component of such a network, in parallel with machines. Then we should in any case distinguish "top level domains", which can only be put in parallel with other domains of the same kind, or with passive entities like messages en route for some destination domain. (This is what is done in NOMADIC PICT for instance.) This distinction is made for instance in calculi of domains with a *flat* structure, à la Dπ (see D1.1.1 for other instances), but it is usually *not* made in calculi with a *hierarchic* organization of domains, like the Mobile Ambients calculus. In this note we examine the two possibilities, flat or hierarchically structured, for a migration calculus.

The migration calculus will only deal with the mobility aspect. This means that the only feature we will be interested in here is the "*permeability*" of the membranes. More precisely, the calculus will only deal with two actions, one for entering and the other for exiting from a domain. In order to do that, the idea is to provide a domain, or rather its membrane, with some means to control its own permeability. That is, *the membrane itself will possess some computing ability*. We could use here, as suggested in the D1.2.0, the notation of the M-calculus, that is $a(S)[P]$, where S, the *control* part of the domain, may in particular implement the control of the movements from the inside or the outside of the domain. However, what we propose is slightly different from the idea of the M-calculus: we do not intend to formalize a calculus where we have a full control over the content of a domain, like with the "passivate" construct of the M-calculus. In fact, what is proposed here can be seen as a simplified M-calculus. We therefore use a different notation. We could use $a\{\!\{ S [P]\!\}$, trying to suggest that it is only the membrane which is equipped with some control S. This is similar to the idea of an "airlock" in the early CHAM. However, this turns out to be not very readable, and we will rather write $a\{S\}[P]$. Here, as in the case of "simple" domains, without any control, part of the meaning of this construct is expressed by the fact that it lets its components compute, that is:

$$\frac{S \to S'}{a\{S\}[P] \to a\{S'\}[P]} \qquad \frac{P \to P'}{a\{S\}[P] \to a\{S\}[P']}$$

The next step is to describe the actions for entering and exiting a domain. However, this depends on the topology of domains we are considering, and we therefore examine separately the two cases.

2 Migration in a flat network

As noted above, in a flat model, with only "top level domains", we can only find domains, and possibly "packets", at the network level. Therefore a process cannot just exit a domain and go outside: it must have some destination, which is another domain (otherwise this would mean that the "ether" has enough computing power to let processes run, which is an assumption that we do not make). More precisely, an exiting process must be transformed into a packet towards another domain. The content of a packet may be something else than a process; it may be for instance a marshalled – or "serialized"([1]) – version $`P$ of a process P, or a tuple made of several components. At a very abstract level, we may assume that the content of a packet is a *message* which we do not yet further analyse, and that a packet is thus of the form $a\langle M\rangle$ where a is the name of the destination domain, and M the message to deliver. Then, assuming given a set of domain names, ranged over by $a, b, c\ldots$, the syntax for (flat) networks of domains is

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

where $a\langle M\rangle$ is a message with destination a, $(A \parallel B)$ is the parallel composition, intended to represent the (physical or logical) juxtaposition of domains, and $(\nu a)A$ is, as in the π-calculus, scope restriction, making the name a private to the sub-network A. In the following we shall describe precisely the operational semantics of these constructs. But let us first discuss the migration features.

[1] A more appropriate terminology would be "gödelized".

As we said, we are mainly concerned with the actions of entering and exiting a domain. Then we assume two constant procedures, **go_in** and **go_out**, the semantics of which is that, inside the membrane of some domain, they cause their argument respectively to enter into the content of the domain – the P part – and to exit from the domain, with a given destination. This can be formulated by the following two schematic rules:

$$\begin{aligned} a\{\cdots \textbf{go\_in}\langle P\rangle \cdots\}[\cdots] &\rightarrow a\{\cdots\}[\cdots P \cdots] \\ a\{\cdots \textbf{go\_out}\langle b, M\rangle \cdots\}[\cdots] &\rightarrow b\langle M\rangle \parallel a\{\cdots\}[\cdots] \end{aligned}$$

Some remarks: when we write $\cdots \textbf{go\_in}\langle P\rangle \cdots$, we mean that the call to the **go_in** procedure, with argument P, is in a position where it can be executed. Similarly, by $\cdots P \cdots$ we mean that the argument P may be added in some way to the content of the domain. For instance, P could be added concurrently to the content of the domain, as a new thread of computation to execute. Then an obvious requirement about the underlying computing model is that it should allow for the dynamic addition of computing entities. Using the CCS paradigm, where each entity, be it a program, a register or a Turing machine's tape, is regarded as a process, and each system, like a memory for instance, is built by juxtaposition, i.e. parallel composition of processes, we would write the previous rules, using parallel composition, as

$$\begin{aligned} a\{\textbf{go\_in}\langle P\rangle \mid S\}[Q] &\rightarrow a\{S\}[P \mid Q] \\ a\{\textbf{go\_out}\langle b, M\rangle \mid S\}[Q] &\rightarrow b\langle M\rangle \parallel a\{S\}[Q] \end{aligned}$$

We shall indeed introduce some constructs, like parallel composition, for the control part S of a domain. However, in order to make as few assumptions as possible regarding the syntax of the "process" part of the model, that is the content P of domains $a\{S\}[P]$, we shall rather make the following hypothesis:

(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 P to an existing content Q, resulting in a process Q', by a labelled transition

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

Then the rules above can be written

$$\frac{Q \xrightarrow{\downarrow P} Q'}{a\{\textbf{go\_in}\langle P\rangle \mid S\}[Q] \rightarrow a\{S\}[Q']}$$

$$a\{\textbf{go\_out}\langle b, M\rangle \mid S\}[Q] \rightarrow b\langle M\rangle \parallel a\{S\}[Q]$$

By these rules, something can leave the membrane for the inside or the outside of the domain, by calling the **go_in** and **go_out** procedures, and the intention is that this is entirely under the control of the membrane itself. Now we need some means for the converse movements. To this end, we make the following assumption:

(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

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

Similarly, we assume that a membrane has the ability to receive these messages. Then there are two rules governing the behaviour of incoming packets and upward messages:

$$\boxed{\begin{array}{c} a\langle M\rangle \parallel a\{S\}[P] \;\rightarrow\; a\{M \mid S\}[P] \\[2ex] \dfrac{P \xrightarrow{\uparrow M} P'}{a\{S\}[P] \;\rightarrow\; a\{S \mid M\}[P']} \end{array}}$$

As one can see, there is no restriction on messages to enter into the membranes, either from the inside or the outside of the domains (but notice that only messages can enter, and these are supposed to be passive entities, that do not compute by themselves). Dealing with the incoming or outcoming messages is indeed one of the main purposes of having a control part in the membrane. On the other hand, there is no direct way for a message to go from the inside to the outside of a domain, or conversely. For this, the procedures **go_in** and **go_out**, which are only known by membranes, must be used. In particular, this means that there is no direct communication from [the content of] a domain to another: here we adhere to the "locality principle", and more precisely to the Dπ "go and communicate" philosophy. One may observe that the following transition occurs, in two steps:

$$a\{\mathbf{go\_out}\langle b, M\rangle \mid S\}[Q] \parallel b\{R\}[P] \;\xrightarrow{*}\; a\{S\}[Q] \parallel b\{M \mid R\}[P]$$

We could have adopted this as an atomic transition, thus doing without explicit packets $b\langle M\rangle$. However, this would violate the locality principle, by which "any computing action should involve at most one domain at a time". Indeed, considering the sequence above as an atomic step would introduce a synchronisation between two domains, whereas by our rules a message is allowed to go out of a locality even if its destination domain does not exist. 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 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

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

(assuming as usual that parallel composition is associative and commutative), 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 $\mathbf{go\_out}\langle a, M\rangle$ 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, which can be expressed with a new construct:

$$\boxed{a\{\mathbf{mk\_dom}\langle b, S, P\rangle \mid T\}[Q] \;\rightarrow\; (\nu b)(b\{S\}[P] \parallel a\{T\}[Q])}$$

where we assume that $b \neq a$. Notice that the name of the created domain is restricted, and therefore does not conflict with existing domain names. In order for this rule to be applicable, we have to assume that the

construct $\mathbf{mk\_dom}\langle b,S,P\rangle$ is a binder for the name b, the scope of which is the domain containing it. We therefore have the following α-conversion rule:

$$a\{\mathbf{mk\_dom}\langle b,S,P\rangle \mid T\}[Q] \equiv_\alpha a\{\mathbf{mk\_dom}\langle c,[c/b]S,[c/b]P\rangle \mid [c/b]T\}[[c/b]Q]$$

where c does not occur free in $a\{\mathbf{mk\_dom}\langle b,S,P\rangle \mid T\}[Q]$.

One can notice that, although we introduced means for a message to go in and out of a membrane, there is still no way for a migrating entity to cross the membrane and go from the outside of a domain into its content, or conversely, because there is no way for a message M to be interpreted inside a membrane. In the following we therefore introduce a particular instance of the migration calculus, where the computing model for the membrane is the one of the π-calculus. It should be easy to design a similar, LINDA-based model, 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 to deal with dynamically incoming function calls), or maybe an object-oriented model, where messages are method invocations. Before introducing any particular instance of it, let us first summarize the basic calculus we have for controlling migration in a flat network. First, the syntax is:

$A, B\ldots$	::=	$a\{S\}[P] \mid a\langle M\rangle \mid (A \parallel B) \mid (\nu a)A$	*networks*
$S, T\ldots$	::=	$\mathbf{go\_in}\langle P\rangle \mid \mathbf{go\_out}\langle b,M\rangle \mid \mathbf{mk\_dom}\langle b,S,P\rangle$	*controllers*
	$\mid$	$M \mid (S \mid T) \mid (\nu a)S$	
$P, Q\ldots$	::=	$\cdots$	*processes*
M	::=	$\cdots$	*messages*

There is no assumption regarding the syntax of messages and processes, but one should remember that we do have some requirements regarding the semantics of processes, see the points **(1)** and **(2)** above. As usual, to formulate the operationnal semantics, we introduce a *structural equivalence*, which is the least equivalence $\equiv$, respectively on networks and controllers, containing the α-conversion relation $\equiv_\alpha$, and such that

$$A \equiv A' \Rightarrow (A \parallel B) \equiv (A' \parallel B) \qquad S \equiv S' \Rightarrow (S \mid T) \equiv (S' \mid T)$$
$$A \equiv A' \Rightarrow (\nu a)A \equiv (\nu a)A' \qquad S \equiv S' \Rightarrow (\nu a)S \equiv (\nu a)S'$$
$$((A \parallel B) \parallel C) \equiv (A \parallel (B \parallel C)) \qquad ((S \mid T) \mid U) \equiv (S \mid (T \mid U))$$
$$(A \parallel B) \equiv (B \parallel A) \qquad (S \mid T) \equiv (T \mid S)$$
$$((\nu a)A \parallel B) \equiv (\nu a)(A \parallel B) \qquad ((\nu a)S \mid T) \equiv (\nu a)(S \mid T) \qquad a \text{ not free in } B \text{ and } T$$

and

$$S \equiv S' \Rightarrow a\{S\}[P] \equiv a\{S'\}[P]$$

Finally the transition rules are

$$a\{\mathbf{go\_out}\langle b,M\rangle \mid S\}[Q] \rightarrow b\langle M\rangle \parallel a\{S\}[Q] \qquad a\langle M\rangle \parallel a\{S\}[P] \rightarrow a\{M \mid S\}[P]$$

$$\frac{Q \xrightarrow{\downarrow P} Q'}{a\{\mathbf{go\_in}\langle P\rangle \mid S\}[Q] \rightarrow a\{S\}[Q']} \qquad \frac{P \xrightarrow{\uparrow M} P'}{a\{S\}[P] \rightarrow a\{S \mid M\}[P']}$$

$$\frac{A \rightarrow A'}{(A \parallel B) \rightarrow (A' \parallel B)} \qquad \frac{A \rightarrow A'}{(\nu a)A \rightarrow (\nu a)A'}$$

$$\frac{S \rightarrow S'}{(S \mid T) \rightarrow (S' \mid T)} \qquad \frac{S \rightarrow S'}{(\nu a)S \rightarrow (\nu a)S'}$$

$$\frac{S \rightarrow S'}{a\{S\}[P] \rightarrow a\{S'\}[P]} \qquad \frac{P \rightarrow P'}{a\{S\}[P] \rightarrow a\{S\}[P']}$$

and

$$a\{\mathbf{mk\_dom}\langle b,S,P\rangle \mid T\}[Q] \;\rightarrow\; (\nu b)(b\{S\}[P] \parallel a\{T\}[Q])$$

with

$$\frac{A \equiv B \qquad A \rightarrow A'}{B \rightarrow A'} \qquad\qquad \frac{S \equiv T \qquad S \rightarrow S'}{T \rightarrow S'}$$

As one can see, a network has no computing ability by itself: it can only transmit (thanks to the associativity and commutativity of parallel composition) messages from domains to domains. Otherwise, computations in a network only occur inside nodes, that is, domains. Notice also that a process P is not allowed to compute inside a call $\mathbf{go\_in}\langle P\rangle$ or $\mathbf{mk\_dom}\langle b,S,P\rangle$.

3 A π-calculus based model

We assume given a set of *names*, ranged over by u, v, $w\ldots$, which in particular contains a set of *variables* x, y, $z\ldots$, but *does not contain* the constants **go_in**, **go_out** and **mk_dom**. We will not make any distinction between process variables, which stand for processes, and channel variables, which stand for channel names (a type system would be useful for that purpose). We also assume that the set of names contains the names a, b, $c\ldots$ we used for domains. The syntax for control programs, occurring in the membranes, is enriched with the standard π-calculus syntax – where some variables may stand for "processes", in the sense of "part of the content of a domain":

$A, B\ldots$	::=	$\cdots \mid (\nu u)A$	*networks*
$S, T\ldots$	::=	$\cdots \mid \mathbf{nil} \mid u(\vec{x}).S \mid \mathbf{!}u(\vec{x}).S \mid (\nu u)S$	*controllers*
M	::=	$u\langle\vec{V}\rangle$	*messages*
V	::=	$u \mid P \mid \cdots$	*values*

The messages are tuples of values sent to 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...). In a LINDA-based model, messages would be $\mathbf{out}(u,t)$, to put tuples (u,t) in the tuple space of the membrane, where the name u is used to identify, by pattern matching, which procedure is actually invoked.

The operational semantics of the extended calculus is standard: besides the rules previously given, we have to extend the rules of scope extrusion in the definition of structural equivalence to the channel names (and possibly to add a rule stating that **nil** is a unit for parallel composition, although this is not necessary for the operational semantics). We also have to add the communication rules:

$$(u\langle\vec{V}\rangle \mid u(\vec{x}).S) \rightarrow [\vec{V}/\vec{x}]S \qquad\qquad (u\langle\vec{V}\rangle \mid \mathbf{!}u(\vec{x}).S) \rightarrow ([\vec{V}/\vec{x}]S \mid \mathbf{!}u(\vec{x}).S)$$

Now let us see some examples. In the π_{1l} and Dπ calculi (see D1.1.1) there is a construction, denoted $\mathbf{go}(b,P)$([2]), to send the process P for execution in the remote site b. Then this can be interpreted as an upward message $^{\uparrow}exit\langle b,P\rangle$ to the enclosing membrane (where *exit* is a name, not a constant of the model), which is supposed to define a local protocol for sending a process elsewhere. Then, assuming that the procedure to enter a domain is called *enter*, a Dπ domain can be represented([3]) as a membrane containing

[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 prefi x $\mathbf{go}\,b$.

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

the following definitions for the ports *enter* and *exit*:

$$(\nu\, exit)a\{\,!enter(x).\mathbf{go\_in}\langle x\rangle \\ \mid !exit(y,x).\mathbf{go\_out}\langle y, enter\langle x\rangle\rangle \\ \}[(\nu\, enter)\ \cdots\ ]$$

Here we have assumed that we are allowed to use name scoping in the process part, in order to make the name *enter* only visible to the outside of a domain – and to its membrane, obviously. Similarly, the restriction $(\nu\, exit)$ has the effect that the name *exit* is not known outside the domain. This may be called a *transparent* membrane, since it does not perform any control on the migrating entities, and just lets them go. In a variant of Dπ recently proposed (and presented in the MIKADO-MYTHS meeting in Brighton), called SAFEDπ, the **go** action has another parameter, which is a higher-order port of the destination locality. This is similar to the *enter* port we used above.

In π_{1l}, 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 membrane of the following kind – assuming that some further computing constructs, like conditional branching, are available:

$$(\nu\, exit)a\{(\nu s)(s\langle \mathbf{true}\rangle \\ \mid !enter(x).\mathbf{go\_in}\langle x\rangle \\ \mid !exit(y,x).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{go\_out}\langle y, enter\langle x\rangle\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid s\langle t\rangle) \\ \mid !stop.s(t).s\langle \mathbf{false}\rangle \\ \mid !ping(y,z).s(t).(\mathbf{go\_out}\langle y, z\langle t\rangle\rangle \mid s\langle t\rangle)\ ) \\ \}[(\nu\, enter)\ \cdots\ ]$$

We could add another restriction $(\nu\, stop)$ in order to deny the ability to stop the domain to the controlled process. 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]). The syntax for the *ping* construct is slightly different from the one of π_{1l}, because this 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 this 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

$$!enter(x).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{go\_in}\langle x\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid s\langle t\rangle)$$

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

$$!enter(x).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{go\_in}\langle x\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{go\_out}\langle a, enter\langle x\rangle\rangle \mid s\langle t\rangle)$$

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 router node r:

$$!exit(y,x).\mathbf{go\_out}\langle r, route\langle y,x\rangle\rangle$$

In the membrane of the site r, the *route* procedure may be defined as follows:

$$!route(y,x).\mathbf{if}\ y = r\ \mathbf{then\ go\_in}\langle x\rangle\ \mathbf{else\ let}\ z = next\_hop\langle y\rangle\ \mathbf{in\ go\_out}\langle z, route\langle y,x\rangle\rangle$$

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

Here we also assume a routing table, called *next_hop*, is available in r, 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:

$$!enter(g,x).\textbf{if } p\langle g\rangle \textbf{ then } \textbf{go\_in}\langle x\rangle \textbf{ else nil}$$

Another example of a membrane in which the enter/exit protocol depends on the local state of the membrane is the *counting* membrane of the Internal Note of April 10. 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:

$$\begin{array}{l}(\nu\, exit)a\{(\nu\, c)(c\langle 0\rangle \\ \qquad\qquad \mid !enter(x).c(z).\textbf{if } z<n \textbf{ then } (\textbf{go\_in}\langle x\rangle \mid c\langle z+1\rangle) \textbf{ else } (\textbf{nil} \mid c\langle z\rangle) \\ \qquad\qquad \mid !exit(y,x).c(z).(\textbf{go\_out}\langle y, enter\langle x\rangle\rangle \mid c\langle z-1\rangle)\ ) \\ \qquad \}[(\nu\, enter)\ \cdots\ ]\end{array}$$

Clearly, the membranes of a domain should be given more computing power. 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 $`P$ of the code of the incoming processes into some data structure. In this case, the **go_out** procedure, sending packets outside, should involve a transformation (marshalling) of its arguments, producing data in a format that can be directly exploited by the membrane of the receiving site.

4 Migration in a hierarchically structured network

4.1 Domains in membranes

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. Since our migration model is as parametric as possible with respect to the computing model used in the content of a domain([5]), it is natural to examine the case where embedded domains can occur within the membrane. In this case, a message exiting an embedded domain can either be addressed to a sibling domain, as in the case of a flat network, or to the membrane where the embedded domain resides([6]). In the latter case the message does not have to mention a destination. Then the syntax for networks in the hierachically structured domains, is actually a part of the syntax for the controllers, namely:

$$S,\, T \ldots \ ::=\ a\{S\}[P] \ \mid\ M \ \mid\ (S \mid T) \ \mid\ \cdots$$

Regarding the operational semantics, only the rule for going out of a membrane has to be changed:

$$\boxed{a\{\textbf{go\_out}\langle M\rangle \mid S\}[Q] \ \rightarrow\ M \mid a\{S\}[Q]}$$

[5] As we suggested above, the controller of a domain should be able to manipulate the code that enters or exits the content part, but there are not necessarily interactions between the two parts otherwise.

[6] considering that there is an anonymous "global domain" that encloses everything.

The three other rules remain the same([7]), but obviously we have to enrich the syntax of messages, in order to be able to cross several membranes in sequence. In particular, a packet $a\langle M\rangle$ should now be considered as a message, so that we have for instance

$$\begin{aligned} a\{\mathbf{go\_out}\langle b\langle M\rangle\rangle \mid S\}[Q] \mid b\{R\}[P] &\rightarrow b\langle M\rangle \mid a\{S\}[Q] \mid b\{R\}[P] \\ &\rightarrow a\{S\}[Q] \mid b\{M \mid R\}[P] \end{aligned}$$

or

$$\begin{aligned} a\langle b\langle M\rangle\rangle \mid a\{b\{S\}[P] \mid R\}[Q] &\rightarrow a\{b\langle M\rangle \mid b\{S\}[P] \mid R\}[Q] \\ &\rightarrow a\{b\{M \mid S\}[P] \mid R\}[Q] \end{aligned}$$

We are assuming here that the membrane still controls only the movements between the inside and the outside of a domain, but not the movements inside the membrane itself.

The syntax of the flat π-calculus based migration model is easily adapted to this hierarchically structured case: as we said, we just have to embed domains into the syntax of controllers, and to extend the syntax for messages.

$$\begin{array}{rcll} S,\, T\ldots & ::= & \mathbf{go\_in}\langle P\rangle \mid \mathbf{go\_out}\langle b, M\rangle \mid \mathbf{mk\_dom}\langle b, S, P\rangle & \textit{controllers/networks} \\ & \mid & a\{S\}[P] \mid M \mid (S \mid T) \mid (\nu a)S & \\ & \mid & \mathbf{nil} \mid u(\vec{x}).S \mid \mathbf{!}u(\vec{x}).S \mid (\nu u)S & \\ M & ::= & u\langle\vec{V}\rangle \mid a\langle M\rangle & \textit{messages} \end{array}$$

It is clear that, if we identify $(A \parallel B)$ with $(A \mid B)$ and $\mathbf{go\_out}\langle b, M\rangle$ with $\mathbf{go\_out}\langle b\langle M\rangle\rangle$, the "flat" migration calculus is a sub-calculus of the hierarchically structured one. However, a question that should be asked is whether we really need embedded domains of this kind, or if we could define an encoding of the full calculus (considering that there is an anonymous "global domain") into the flat one. One should notice that, in the calculus we just sketched, domains are immobile: once created, they cannot be moved. We could perhaps accomodate the subjective moves of domains, as suggested in the Internal Note of April 10, but it is not clear that it is worth doing so. (There is no way to dissolve a domain in the calculus, and there seems to be no meaningful way of doing that, unless the computing model for processes is the same as the one of the domain controllers.)

4.2 Domains in processes

There is obviously another way to build hierarchically structured domains, which is to consider that domains are embedded in processes. This is perhaps more interesting than the previous notion of a hierarchical structure, since we now have the ability to control domains, from the membranes. However, it is clear that we cannot in this case maintain 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:

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

In this case, we may represent the dynamic addition of new processes using parallel composition:

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

Similarly, we may also assume that upward messages are part of the process syntax, with

$$({}^{\uparrow}M \mid P) \xrightarrow{\uparrow M} P$$

[7] except that we should use "ordinary" parallel composition in the left-hand side of the rule for incoming messages.

There is nothing to change in the semantics of controllers. On the other hand, the semantics of processes should now include the rules we gave where networks are involved (using "ordinary" parallel composition $|$ instead of $\|$).

In a π-calculus based instantiation of this hierarchically structured model of domains, it is actually natural to merge the two categories of networks and processes into just one, thus obtaining the following syntax:

$$
\begin{array}{rcll}
P, Q\ldots & ::= & a\{S\}[P] \mid a\langle M\rangle \mid {}^{\uparrow}M \mid (P \mid Q) \mid (\nu a)P \mid \cdots & \textit{networks of processes} \\
S, T\ldots & ::= & \mathbf{go\_in}\langle P\rangle \mid \mathbf{go\_out}\langle b, M\rangle \mid \mathbf{mk\_dom}\langle b, S, P\rangle & \textit{controllers} \\
 & \mid & M \mid (S \mid T) \mid (\nu a)S & \\
 & \mid & \mathbf{nil} \mid u(\vec{x}).S \mid \mathbf{!}u(\vec{x}).S \mid (\nu u)S & \\
M & ::= & u\langle \vec{V}\rangle & \textit{messages} \\
V & ::= & u \mid P \mid \cdots & \textit{values}
\end{array}
$$

The operational semantics is, mutatis mutandis (and in particular using $|$ instead of $\|$) and interpreting the transitions $Q \xrightarrow{{}^{\downarrow}P} Q'$ and $P \xrightarrow{{}^{\uparrow}M} P'$ as indicated above, the same as the one of the "flat" π-calculus based model, which we gave in Sections 2 and 3. Therefore we have the same examples of membranes as the ones of Section 3. Since here a domain is a process, we gain the ability of moving domains around (in an "objective" manner, that is as the content of messages). 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.