Security Policies as Membranes in Systems for Global Computing [1]

Daniele Gorla [2]

Dip. di Informatica, Univ. di Roma "La Sapienza", Italy
Dip. di Sistemi ed Informatica, Univ. di Firenze, Italy

Matthew Hennessy [3]

Dep. of Informatics, Univ. of Sussex, Brighton (UK)

Vladimiro Sassone [4]

Dep. of Informatics, Univ. of Sussex, Brighton (UK)

Abstract

We propose a simple global computing framework, whose main concern is code migration. Systems are structured in sites, and each site is divided into two parts: a computing body, and a *membrane* which regulates the interactions between the computing body and the external environment. More precisely, membranes are filters which control access to the associated site, and they also rely on the well-established notion of *trust* between sites. We develop a basic theory to express and enforce security policies via membranes. Initially, these only control the actions incoming agents intend to perform locally. We then adapt the basic theory to encompass more sophisticated policies, where the number of actions an agent wants to perform, and also their order, are considered.

Key words: Global Computing, Code Migration, Access Control, Security Policies, Types.

[1] This work has been partially supported by EU FET – Global Computing initiative, projects MIKADO IST-2001-32222 and MyThS IST-2001-32617. The funding bodies are not responsible for any use that might be made of the results presented here.

[2] Email: gorla@di.uniroma1.it

[3] Email: matthewh@susx.ac.uk

[4] Email: vs@susx.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Computing is increasingly characterised by the global scale of applications and the ubiquity of interactions between mobile components. Among the main features of the forthcoming "global ubiquitous computing" paradigm we list *distribution* and *location awarness*, whereby code located at specific sites acts appropriately to local parameters and circumstances, that is, it is "context-aware"; *mobility*, whereby code is dispatched from site to site to increase flexibility and expressivity; *openness*, reflecting the nature of global networks and embodying the permeating hypothesis of localised, partial knowledge of the execution environment. Such systems present enormous difficulties, both technical and conceptual, and are currently more at the stage of exciting future prospectives than that of established of engineering practice. Two concerns, however, appear to clearly have a ever-reaching import: *security* and *mobility control*, arising respectively from openness and from massive code and resource migrations. They are the focus of the present paper.

We aim at classifying mobile components according to their behaviour, and at empowering sites with control capabilities which allow them to deny access to those agents whose behaviour does not conform to the site's *policy*. We see every site of a system

$$k[\![\, M \,\rangle\!\rangle\, P \,]\!]$$

as an entity named k and structured in two layers: a *computing body* P, where programs run their code – possibly accessing local resources offered by the site – and a *membrane* M, which regulates the interactions between the computing body and the external environment. An agent P wishing to enter a site N must be verified by the membrane before it is given a chance to execute in N. If the preliminary check succeeds, the agent is allowed to execute, otherwise it is rejected. In other words, a membrane implements the policy each site wants to enforce locally, by ruling on the requests of access of the incoming agents. This can be easily expressed by a migration rule of the form:

$$k[\![\, M^k \,\rangle\!\rangle\, \mathbf{go}l.P \mid Q \,]\!] \parallel l[\![\, M^l \,\rangle\!\rangle\, R \,]\!] \to k[\![\, M^k \,\rangle\!\rangle\, Q \,]\!] \;\parallel\; l[\![\, M^l \,\rangle\!\rangle\, P \mid R \,]\!] \qquad \text{if } M^l \vdash^k P$$

The relevant parts here are P, the agent wishing to migrate from k to l, and l, the receiving site, which needs to be satisfied that P's behaviour complies with its policy. The latter is expressed by l's membrane, M^l. The judgement $M^l \vdash^k P$ represents l inspecting the incoming code to verify that it upholds M^l.

Observe that in the formulation above $M^l \vdash^k P$ represent a runtime check of all incoming agents. Because of our fundamental assumption of openendedness, such kind of checks, undesirable as they, cannot be avoided. In order to reduce their impact on systems performance, and to make the runtime semantics as efficient as

possible, we adopt a strategy which allows for efficient agent verification. Precisely, we adopt an elementary notion of *trust*, so that from the point of view of each l the set of sites is consistently partitioned between "good," "bad," and "unknown" sites. Then, in a situation like the one in the rule above, we assume that l will be willing to accept from a *trusted* site k a *k-certified digest* T of P's behaviour. We then modify the primitive **go** and the judgement $\vdash^k$ as in the refined migration rule below.

$$k[\![\, M^k \,\rangle\!\rangle\, \mathbf{go}_{\mathsf{T}} l.P \mid Q \,]\!] \parallel l[\![\, M^l \,\rangle\!\rangle\, R \,]\!] \rightarrow k[\![\, M^k \,\rangle\!\rangle\, Q \,]\!] \;\parallel\; l[\![\, M^l \,\rangle\!\rangle\, P \mid R \,]\!] \qquad \text{if } M^l \vdash^k_{\mathsf{T}} P$$

The notable difference is in $M^l \vdash^k_{\mathsf{T}} P$. Here, l verifies the entire code P against M^l *only if* it does not trust k, the signer of P's certificate T. Otherwise, it suffices for l to match M^l against the digest T carried by **go** together with P from k, so effectively shifting work from l to the originator of P.

Our main concern in this paper is to put the focus on the machinery a membrane should implement to enforce *different kinds* of policies. We first distill the simplest calculus which can conceivably convey our ideas and still support a non-trivial study. It is important to remark that we are abstracting from agents' local computations. These can be expressed in any of several well-known models for concurrency, for example CCS [12] or the π–calculus [13]. We are concerned, instead, with agents' migration from site to site: our main language mechanism is **go** rather than intra-site (i.e. local) communication. Using this language, we examine four notions of policy and show how they can be enforced by using membranes. We start with an amusingly simple policy which only lists allowed actions. We then move to count action occurrences and then to policies expressed by *deterministic finite automata*. Note that such policies are only concerned with the behaviour of single agents, and do *not* take into account "*coalitional*" behaviours, whereby incoming agents – apparently innocent – join clusters of resident agents – they too apparently innocent – to perform cooperatively potentially harmful actions, or at least overrule the host site's policy. We call *resident* those policies intended to be applied to the joint, composite behaviour of the agents contained at a site. We explore resident policies as our fourth and final notion of policy. In all the cases, the theory adapts smoothly to the various cases; we only need to refine the information stored in the membrane and the inspection mechanisms.

Structure of the paper. In Section 2 we define the calculus used in this paper, and start with the straightforward policy which only prescribes the actions an agent can perform when running in a site. In Section 3, we enhance the theory to control also how many (and not only which kind of) actions an agent wants to perform in a site, and their order of execution. Finally, in Section 4 we extend the theory to control the overall computation taking place at a site, and not only the behaviour of single agents. The paper concludes in Section 5 where a comparison with related work is also given. The theoretical results are proved in the full paper [7].

$$\begin{array}{lrcl} \textit{Agents} & P, Q, R & ::= & \mathbf{nil} \mid a.P \mid \mathbf{go}_{\mathsf{T}} l.P \mid P|Q \mid !P \\ \textit{Systems} & N & ::= & \mathbf{0} \mid l[\![\, M \,\rangle\!\rangle\, P \,]\!] \mid N_1 \parallel N_2 \end{array}$$

Fig. 1. A Simple Calculus

2 A Simple Calculus

In this section we describe a simple calculus for mobile agents, which may migrate between sites. Each site is guarded by a *membrane*, whose task is to ensure that every agent accepted at the site conforms to an *entry policy*.

2.1 The Syntax

The syntax is given in Figure 1 and assumes two pairwise disjoint sets: basic agent actions Act, ranged over by $a, b, c, \cdots$, and localities Loc, ranged over by $l, k, h, \cdots$. Agents are constructed using the standard action-prefixing, parallel composition and replication operators from process calculi, [12]. The one novel operator is that for migration, $\mathbf{go}_{\mathsf{T}} l.P$. This agent seeks to migrate to site l in order to execute the code P; moreover it promises to conform to the entry policy T. In practical terms this might consist of a certification that the incoming code P conforms to the policy T, which the site l has to decide whether or not to accept. In our framework, this certification is a policy that describes the (local) behaviour of the agent; thus, in $\mathbf{go}_{\mathsf{T}} l.P$, T will be called the *digest* of P.

A system consists of a finite set of sites running in parallel. A site takes the form $l[\![\, M \,\rangle\!\rangle\, P \,]\!]$, where l is the site name, P is the code currently running at l, and M is the membrane which implements the entry policy. For convenience we assume that site names are unique in systems. Thus, in a given system we can identify the membrane associated with the site named l by M^l. We start with a very simple kind of policy, which we will then progressively enhance.

Definition 2.1 (Policies) A *policy* is any finite subset of Act $\cup$ Loc. For two policies T_1 and T_2, we write T_1 `enforces` T_2 whenever $\mathsf{T}_1 \subseteq \mathsf{T}_2$. ■

Intuitively an agent conforms to a policy T at a given site if every action it performs at the site is contained in T, and it will only migrate to sites whose names are in T. For example, conforming to the policy {`info`, `req`, HOME}, where `info`, `req` are actions and HOME a location, means that the only actions that will be performed are from the set {`info`, `req`} and migration will only occur, if at all, to

$$\text{(R-ACT)} \qquad l[\![\, M \,\rangle\!\rangle\, a.P \mid Q \,]\!] \;\to\; l[\![\, M \,\rangle\!\rangle\, P \mid Q \,]\!]$$

$$\text{(R-PAR)} \qquad \frac{N_1 \;\to\; N_1'}{N_1 \parallel N_2 \to N_1' \parallel N_2}$$

$$\text{(R-STRUCT)} \qquad \frac{N \equiv N_1 \quad N_1 \to N_1' \quad N_1' \equiv N'}{N \to N'}$$

$$\text{(R-MIG)} \qquad k[\![\, M^k \,\rangle\!\rangle\, \mathbf{go}_{\mathsf{T}} l.P \mid Q \,]\!] \;\parallel\; l[\![\, M^l \,\rangle\!\rangle\, R \,]\!] \;\to\; k[\![\, M^k \,\rangle\!\rangle\, Q \,]\!] \;\parallel\; l[\![\, M^l \,\rangle\!\rangle\, P \mid R \,]\!] \quad \text{if} \;\; M^l \vdash^k_{\mathsf{T}} P$$

Fig. 2. The reduction relation

the site HOME. With this interpretation of policies, our definition of the predicate `enforces` is also intuitive; if some code P conforms to the policy T_1 and T_1 `enforces` T_2 then P also automatically conforms to T_2.

The purpose of membranes is to enforce such policies on incoming agents. In other words, at a site $l[\![\, M \,\rangle\!\rangle\, Q \,]\!]$ wishing to enforce a policy T_{in}, the membrane M has to decide when to allow entry to an agent such as $\mathbf{go}_{\mathsf{T}} l.P$ from another site. There are two possibilities.

- The first is to syntactically check the code P against the policy T_{in}; an implementation would actually expect the agent to arrive with a proof of this fact, and this proof would be checked.
- The second would be to *trust* the agent that its code P conforms to the stated T and therefore only check that this conforms to the entry policy T_{in}. Assuming that checking one policy against another is more efficient than the code analysis, this would make entry formalities much easier.

Deciding on when to apply the second possibility presupposes a *trust management* framework for systems, which is the topic of much current research. To simplify matters, here we simply assume that each site contains, as part of its membrane, a record of the level of trust it has in other sites. Moreover, we assume only three possible levels: `bad`, `unknown` and `good`.

Definition 2.2 (Membranes) A membrane M is a pair $(M_t, \mathsf{M}_{\mathsf{p}})$ where M_t is a partial function from LOC to $\{\mathtt{unknown}, \mathtt{good}, \mathtt{bad}\}$, and M_{p} is a policy. ■

2.2 The Operational Semantics

Having defined both *policies* and *membranes*, we now give an operational semantics for the calculus, which formalises the above discussion of how to manage agent migration. This is given as a binary relation $N \rightarrow N'$ over systems; it is defined to be the least relation which satisfies the rules in Figure 2. Rule (R-ACT) says that the agent $a.P$ running in parallel with other code in site l, such as Q, can perform the action a; note that the semantics does not record the occurrence of a. (R-PAR) and (R-STRUCT) are standard. The first allows reductions within parallel components, while the second says that reductions are relative to a structural equivalence. The precise rules defining this equivalence are unsurprising and therefore left to the full paper [7]; they state that '|' and '‖' are monoidal operators (with **nil** and **0** acting as identities, resp.), and that replicated processes can be freely unfolded. The interesting reduction rule is the last one, (R-MIG), governing migration; the agent $\mathbf{go}_{\mathsf{T}} l.P$ can migrate from site k to site l provided the predicate $M^l \vdash^k_{\mathsf{T}} P$ is true. This 'enabling' predicate formalises our discussion above on the role of the membrane M^l, and requires in turn a notion of code P satisfying a policy T,

$$\vdash P : \mathsf{T}$$

With such a notion, we can then define $M^l \vdash^k_{\mathsf{T}} P$ to be:

$$\textbf{if } M^l_t(k) = \mathtt{good} \textbf{ then } (\mathsf{T} \ \mathtt{enforces} \ \mathsf{M}^l_{\mathsf{p}}) \textbf{ else } \vdash P : \mathsf{M}^l_{\mathsf{p}} \qquad (1)$$

In other words, if the target site l trusts the source site k, it trusts that the professed policy T is a faithful reflection of the behaviour of the incoming agent P, and then entry is gained provided that T enforces the entry policy $\mathsf{M}^l_{\mathsf{p}}$ (i.e., in this case, $\mathsf{T} \subseteq \mathsf{M}^l_{\mathsf{p}}$). Otherwise, if k can not be trusted, then the entire incoming code P has to be checked to ensure that it conforms to the entry policy, as expressed by the predicate $\vdash P : \mathsf{M}^l_{\mathsf{p}}$.

In Figure 3 we describe a simple inference system for checking that agents conform to policies, i.e. to infer judgements of the form $\vdash P : \mathsf{T}$. Rule (TC-EMPTY) simply says that the empty agent **nil** satisfies all policies. (TC-ACT) is also straightforward; $a.P$ satisfies a policy T and if a is allowed by T, and the residual P satisfies T. The rule (TC-PAR) says that to check $P \mid Q$ it is sufficient to check P and Q separately, and similarly for replicated agents. The most interesting rule is (TC-MIG), which checks $\mathbf{go}_{\mathsf{T}'} l.P$. This not only checks that migration to l is allowed by the policy, that is $l \in \mathsf{T}$, but it also checks that the code to be spawned there, P, conforms to the associated professed policy T'. In some sense, if the agent $\mathbf{go}_{\mathsf{T}'} l.P$ is allowed a entry into a site k, then k assumes responsibility for any promises that it makes about conformance to policies.

$$\text{(TC-EMPTY)}\ \vdash \mathbf{nil} : \mathsf{T} \qquad \text{(TC-ACT)}\ \frac{\vdash P : \mathsf{T}}{\vdash a.P : \mathsf{T}}\, a \in \mathsf{T} \qquad \text{(TC-MIG)}\ \frac{\vdash P : \mathsf{T}'}{\vdash \mathbf{go}_{\mathsf{T}'} l.P : \mathsf{T}}\, l \in \mathsf{T} \qquad \text{(TC-REPL)}\ \frac{\vdash P : \mathsf{T}}{\vdash\, !P : \mathsf{T}} \qquad \text{(TC-PAR)}\ \frac{\vdash P : \mathsf{T} \qquad \vdash Q : \mathsf{T}}{\vdash P \mid Q : \mathsf{T}}$$

Fig. 3. Typechecking incoming agents

2.3 Safety

We have just outlined a reduction semantics in which sites seek to enforce policies either by directly checking the code of incoming agents against entry policies, or more simply by checking the professed policy of trusted agents. The extent to which this strategy works depends, not surprisingly, on the quality of a site's trust management.

Example 2.1 Let HOME be a site name with the following trust function

$$M_t^h \ : \ \{\text{ALICE}, \text{BOB}, \text{SECURE}\} \quad \mapsto \quad \texttt{good}\ .$$

Consider the system

$$S \triangleq \text{HOME}[\![\, M^h \,\rangle\!\rangle\, P^h \,]\!] \ \parallel\ \text{BOB}[\![\, M^b \,\rangle\!\rangle\, P^b \,]\!] \ \parallel\ \text{ALICE}[\![\, M^a \,\rangle\!\rangle\, P^a \,]\!] \ \parallel\ \text{SECURE}[\![\, M^s \,\rangle\!\rangle\, P^s \,]\!]$$

in which the entry policy of HOME, M_p^h, is $\{\texttt{info},\ \texttt{req},\ \text{SECURE}\}$, and that of SECURE, M_p^s, is $\{\texttt{give},\ \text{HOME}\}$. Since $M_t^h(\text{BOB}) = \texttt{good}$, agents migrating from BOB to HOME are trusted and only their digests are checked against the entry policy M_p^h. So, if P^b contains the agent

$$\mathbf{go}_{\mathsf{T}_1}\text{HOME}.(\texttt{take}.Q)$$

where T_1 `enforces` M_p^h, then the entry policy of HOME will be transgressed. ■

The problem in this example is that the trust knowledge of HOME is faulty; it trusts in sites which do not properly ensure that professed policies are enforced. Let us divide the sites into *trustworthy* and otherwise. This bipartition could be stored in an external record stating which nodes are trustworthy (i.e. typechecked) and which ones are not. However, for economy, we prefer to record this information in the membranes, by demanding that the trust knowledge at trustworthy sites is a proper reflection of this division. This is more easily defined if we assume the following ordering over trust levels:

$$\texttt{unknown} <: \texttt{bad} \qquad \text{and} \qquad \texttt{unknown} <: \texttt{good}$$

$$\frac{}{\vdash \mathbf{0} : \mathbf{ok}}\ (\text{WF-EMPTY}) \qquad \frac{\vdash P : \mathsf{M_p}}{\vdash l[\![M \rangle P]\!] : \mathbf{ok}}\ (\text{WF-G.SITE})\ l \text{ trustworthy}$$

$$\frac{\vdash N_1 : \mathbf{ok}, \qquad \vdash N_2 : \mathbf{ok}}{\vdash N_1 \ \| \ N_2 : \mathbf{ok}}\ (\text{WF-PAR}) \qquad \frac{}{\vdash l[\![M \rangle P]\!] : \mathbf{ok}}\ (\text{WF-U.SITE})\ l \text{ not trustworthy}$$

Fig. 4. Well-formed systems

This reflects the intuitive idea that sites classified as `unknown` may, perhaps with further information, be subsequently classified either as `good` or `bad`. On the other hand, `good` or `bad` cannot be further refined; sites classified as either, will not be reclassified.

Definition 2.3 (Trustworthy sites and Coherent systems) In a system N, we say the site k is *trustworthy* if $M_t^k(k) = \texttt{good}$. N is *coherent* if for every trustworthy site k, it holds that $M_t^k(l) <: M_t^l(l)$. ■

Thus, if a trustworthy site k believes that a site l can be trusted (i.e., $M_t^k(l) = \texttt{good}$), then l is indeed trustworthy (as represented by $M_t^l(l) = \texttt{good}$). Similarly, if it believes l to be `bad`, then l is indeed bad. The only uncertainty is when k classifies l as `unknown`: then l may be either `good` or `bad`. Of course, in coherent systems we expect sites which have been classified as trustworthy to act in a *trustworthy manner*, which amounts to say that code running at such a k must have at one time gained entry there by satisfying the entry policy. Note that by using policies as in Definition 2.1, if P satisfies an entry policy $\mathsf{M_p^k}$, then it continues to satisfy the policy while running at k (cf. Theorem 2.2 below).

This property of coherent systems, which we call *well-formedness*, can therefore be checked syntactically. In Figure 4, we give the set of rules for deriving the judgement $\vdash N : \mathbf{ok}$, of well-formedness of N. There are only two interesting rules. Firstly, (WF-G.SITE) says that $l[\![M \rangle P]\!]$ is well-formed whenever l is trustworthy and $\vdash P : \mathsf{M_p}$. There is a subtlety here; this not only means that P conforms to the policy $\mathsf{M_p}$, but also that any digests proffered by agents in P can also be trusted. The second relevant rule is (WF-U.SITE), for typing unknown sites: here there is no need to check the resident code, as agents emigrating from such sites will not be trusted.

Example 2.2 (*Example 2.1 continued.*) Let us now re-examine the system S in Example 2.1. Suppose HOME is trustworthy, that is $M_t^h(\text{HOME}) = \texttt{good}$. Then, if S is to be coherent, it is necessary for each of the sites BOB, ALICE and SECURE also

$$\text{(LTS-ACT)}\quad a.P \xrightarrow{a} P \qquad \text{(LTS-MIG)}\quad \mathbf{go}_{\mathsf{T}} l.P \xrightarrow{l} \mathbf{nil}$$

$$\text{(LTS-REPL)}\quad \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \qquad \text{(LTS-PAR)}\quad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2 \quad P_2 \mid P_1 \xrightarrow{\alpha} P_2 \mid P_1'}$$

Fig. 5. A Labelled Transition System

to be trustworthy. Consequently, S can not be well-formed. For example, to derive $\vdash S : \mathbf{ok}$ it would be necessary to derive

$$\vdash \mathbf{go}_{\mathsf{T}_1}\text{HOME}.(\mathtt{take}.Q) : \mathsf{M}_{\mathsf{p}}^b$$

where $\mathsf{M}_{\mathsf{p}}^b$ is the entry policy of BOB. But this requires the judgement $\vdash \mathtt{take}.Q : \mathsf{T}_1$, where T_1 $\mathtt{enforces}$ $\mathsf{M}_{\mathsf{p}}^h$. Since $\mathtt{take} \notin \mathsf{M}_{\mathsf{p}}^h$, this is not possible. ■

In well-formed systems we know that entry policies have been respected. So one way of demonstrating that our reduction strategy correctly enforces these policies is to prove two things: system well-formedness is preserved by reduction, and only legal computations take place within trustworthy sites. The first requirement is straightforward to formalize:

Theorem 2.1 (Subject Reduction) *If* $\vdash N : \mathbf{ok}$ *and* $N \to N'$, *then* $\vdash N' : \mathbf{ok}$.

To formalise the second requirement we need some notion of the *computations* of an agent. With this in mind, we first define a labelled transition system between agents, which details the immediate actions an agent can perform, and the residual of those actions. The rules for the judgements $P \xrightarrow{\alpha} Q$, where we let α to range over $\text{Act} \cup \text{Loc}$, are given in Figure 5, and are all straightforward. These judgements are then extended to $P \xrightarrow{\sigma} Q$, where σ ranges over $(\text{Act} \cup \text{Loc})^*$, in the standard manner: $\sigma = \alpha_1, \ldots, \alpha_k$, when there exists $P_0, \ldots, P_k$ such that $P = P_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_k} P_k = P'$. Finally, let $\mathtt{act}(\sigma)$ denote the set of all elements of $\text{Act} \cup \text{Loc}$ in σ.

Theorem 2.2 (Safety) *Let* N *be a well-formed system. Then, for every trustworthy site* $l[\![M \rangle P]\!]$ *in* N, $P \xrightarrow{\sigma} P'$ *implies that* $\mathtt{act}(\sigma)$ $\mathtt{enforces}$ M_{p}.

3 Entry Policies

The calculus of the previous section is based on a simple notion of entry policies, namely finite sets of actions and location names. An agent conforms to such a policy T at a site if it only executes actions in T before migrating to some location in T. However both the syntax and the semantics of the calculus are completely parametric on policies. All that is required of the collection of policies is a binary relation T_1 $\mathtt{enforces}$ T_2 between them, and a binary relation $\vdash P : \mathsf{T}$ indicating that the code P conforms to the policy T. With any collection of policies, endowed

with two such relations, we can define the predicate $M \vdash^k_{\mathsf{T}} P$ as in (1) above, and thereby get a reduction semantics for the calculus. In this section we investigate two variations on the notion of entry policies and discuss the extent to which we can prove that the reduction strategy correctly implements them.

3.1 Multisets as Entry Policies

The policies of the previous section only express the legal actions agents may perform at a site. However in many situations more restrictive policies are desirable. To clarify this point, consider the following example.

Example 3.1 Let MAIL_SERV be the site name of a mail server with the following entry policy $\mathsf{M}_{\mathsf{p}}^{ms}$: $\{\mathtt{list}, \mathtt{send}, \mathtt{retr}, \mathtt{del}, \mathtt{reset}, \mathtt{quit}\}$. The server accepts client agents performing requests for listing mail messages, sending/retrieving/deleting messages, resetting the mailbox and quitting. Now, consider the system

$$S \triangleq \text{MAIL\_SERV}[\![\, M^{ms} \,\rangle\!\rangle\, P^{ms} \,]\!] \;\;\|\;\; \text{SPAM}[\![\, M^{s} \,\rangle\!\rangle\, \mathbf{go}_{\mathsf{T}}\text{MAIL\_SERV}.(!\mathtt{send}) \,]\!]$$

where $\mathsf{T} = \{\mathtt{send}\}$. According to the typechecking of Figure 3, we have that $\vdash !\,\mathtt{send} : \mathsf{M}_{\mathsf{p}}^{ms}$. However, the agent is a spamming virus and, in practical implementations, should be rejected by MAIL_SERV. ■

In such scenarios it would be more suitable for policies to be able to fix an upper-bound over the number of messages sent. This can be achieved in our setting by changing policies from sets of agent actions to *multisets* of actions.

First let us fix some notation. We can view a multiset as a set equipped with an *occurrence function*, that associates a natural number to each element of the set. To model permanent resources, we also allow the occurrence function to associate ω to an element with an infinite number of occurrences in the multiset. Notationally, $\mathtt{e}^{\omega}$ stands for an element $\mathtt{e}$ occurring infinitely many times in a multiset. This notation is extended to sets; for any set E, we let E^{ω} to denote the multiset $\{\mathtt{e}^{\omega} : \mathtt{e} \in E\}$.

Example 3.2 (*Example 3.1 continued.*) Coming back to Example 3.1, it would be sufficient to define $\mathsf{M}_{\mathsf{p}}^{ms}$ to be $\{\ldots, \mathtt{send}^{K}, \ldots\}$ where K is a reasonable constant. In this way, an agent can only send at most K messages in each session; if it wants to send more messages, it has to disconnect from MAIL_SERV (i.e. leave it) and then reconnect again (i.e. immigrate again later on). ■

The theory presented in Sections 2.2 and 2.3 can be adapted to the case where policies are multisets of actions. The judgment $\vdash P : \mathsf{T}$ is redefined in Figure 6, where operator $\cup$ stands for *multiset union*. The key rules are (TC-ACT), (TC-PAR) and (TC-REPL). The first two properly decrease the type satisfied when typechecking sub-agents. The third one is needed because recursive agents can be, in general,

$$\text{(TC-EMPTY)}\quad \vdash \mathbf{nil} : \mathsf{T} \qquad \text{(TC-ACT)}\quad \frac{\vdash P : \mathsf{T}}{\vdash a.P : \mathsf{T} \cup \{\mathsf{a}\}} \qquad \text{(TC-MIG)}\quad \frac{\vdash P : \mathsf{T}'}{\vdash \mathbf{go}_{\mathsf{T}'} l.P : \mathsf{T} \cup \{l\}}$$

$$\text{(TC-PAR)}\quad \frac{\vdash P : \mathsf{T}_1 \qquad \vdash Q : \mathsf{T}_2}{\vdash P \mid Q : \mathsf{T}_1 \cup \mathsf{T}_2} \qquad \text{(TC-REPL)}\quad \frac{\vdash P : \mathsf{T}}{\vdash\, !P : \mathsf{T}'}\ \mathsf{T}^\omega \ \texttt{enforces}\ \mathsf{T}'$$

Fig. 6. Typechecking with policies as Multisets

freely unfolded; hence, the actions they intend to locally perform can be iterated arbitrarily many times. For instance, agent $P \triangleq !\,\texttt{send}$, satisfies policy $\mathsf{T} \triangleq \{\texttt{send}^\omega\}$. Notice that the new policy satisfaction judgement prevents the spamming virus of Example 3.1 from typechecking against the policy of MAIL_SERV defined in Example 3.2.

The analysis of the previous section can also be repeated here but an appropriate notion of *well-formed* system is more difficult to formulate. The basic problem stems from the difference between *entry* policies and *resident* policies. The fact that all agents who have ever entered a site l respects an entry policy $\mathsf{M_p}$ gives no guarantees as to whether the joint effect with the code currently occupying the site l also satisfies $\mathsf{M_p}$. For instance, in the terms of Example 3.2, MAIL_SERV ensures that each incoming agent can only send at most K messages. Nevertheless, two such agents, having gained entry and now running concurrently at MAIL_SERV, can legally send – jointly – up to $2K$ messages. It is therefore necessary to formulate *well-formedness* in terms of the individual threads of the code currently executing at a site. Let us say P is a *thread* if it is not of the form $P_1 \mid P_2$. Note that every agent P can be written in the form of $P_1 | \ldots | P_n, n \geq 1$, where each P_i is a thread. So the well-formedness judgment is modified by replacing rule (WF-G.SITE) in Figure 4 as below.

$$\text{(WF-G.SITE}_\text{M}\text{)}\quad \frac{\forall i\,.\,(P_i \text{ a thread and } \vdash P_i : \mathsf{M_p})}{\vdash l[\![\, M \,\rangle\!\rangle\, P_1 | \ldots | P_n \,]\!] : \mathbf{ok}}\ l \text{ trustworthy}$$

Theorem 3.1 (Subject Reduction for multiset policies) *If* $\vdash N : \mathbf{ok}$ *and* $N \to N'$, *then* $\vdash N' : \mathbf{ok}$.

The statement of safety must be changed to reflect the focus on individual threads rather than agents.

Theorem 3.2 (Safety for multiset policies) *Let* N *be a well-formed system. Then, for every trustworthy site* $l[\![\, M \,\rangle\!\rangle\, P_1 | \ldots | P_n \,]\!]$ *in* N, *where each* P_i *is a thread,* $P_i \xrightarrow{\sigma} P_i'$ *implies that* $\texttt{act}(\sigma)\ \texttt{enforces}\ \mathsf{M_p}$.

3.2 Finite Automata as Entry Policies

A second limitation of the setting presented in Section 2 is that policies will sometimes need to prescribe a precise order for executing legal actions. This is very common in client/server interactions, where a precise protocol (i.e. a pattern of message exchange) must be respected. To this aim, we define policies as *deterministic finite automata* (DFAs, for short).

Example 3.3 Let us consider Example 3.1 again. Usually, mail servers requires a preliminary authentication phase to give access to mail services. To express this fact, we could implement the entry policy of MAIL_SERV, $\mathsf{M}_{\mathsf{p}}^{ms}$, to be the automaton associated to the regular expression below.

$$\texttt{usr.pwd.(list} + \texttt{send} + \texttt{retr} + \texttt{del} + \texttt{reset)}^*\texttt{.quit}$$

The server accepts client requests only upon authentication, via a username/password mechanism. Moreover, the policy imposes that each session is regularly committed by imposing that each sequence of actions is terminated by `quit`. This could be required to save the status of the transaction and avoid inconsistencies. ■

We now give the formal definitions needed to adapt the theory developed in Section 2. We start by defining a DFA, the language associated to it, the `enforces` predicate between DFAs and a way for an agent to satisfy a DFA. As usual [10], a DFA is a quintuple $\mathsf{A} \triangleq (S, \Sigma, s_0, F, \delta)$ where S is a finite set of *states*, Σ is the *input alphabeth*, $s_0 \in S$ is the *starting state*, $\emptyset \subset F \subseteq S$ is the set of *final states*, and $\delta : S \times \Sigma \to S$ is the *transition relation* In our framework, the alphabeth of the DFAs considered is a finite subset of Act ∪ Loc. Moreover, for the sake of simplicity, we shall always assume that the DFAs in this paper are minimal.

Definition 3.1 (DFA Acceptance and Enforcement) Let A be a DFA. Then

- $Acp_s(\mathsf{A})$ contains all the $\sigma \in \Sigma^*$ such that σ leads A from state s to a final state;
- $Acp(\mathsf{A})$ is defined to be $Acp_{s_0}(\mathsf{A})$;
- A_1 `enforces` A_2 holds true whenever $Acp(\mathsf{A}_1) \subseteq Acp(\mathsf{A}_2)$. ■

We now formally describe the language associated to an agent. To this aim, we exploit the notion of *concurrent regular expressions* (CRE, for short) introduced in [6] to model concurrent processes. For our purposes, the following subset of CRE suffices:

$$e \quad ::= \quad \epsilon \quad | \quad \alpha \quad | \quad e_1.e_2 \quad | \quad e_1 \odot e_2 \quad | \quad e^{\otimes}$$

ϵ denotes the empty sequence of characters, α ranges over Act∪Loc, '.' denotes concatenation, $\odot$ is the interleaving (or *shuffle*) operator and $^{\otimes}$ is its closure. Intuitively,

if e represents the language L, then $e^{\otimes}$ represents $\{\epsilon\} \cup L \cup L \odot L \cup L \odot L \odot L \ldots$. Given a CRE e, the language associated to it, written $lang(e)$, can be easily defined; a formal definition is given in the full paper. Now, given a process P, we easily define a CRE associated to it. Formally

$$\mathrm{CRE}(\mathbf{nil}) \triangleq \epsilon \qquad \mathrm{CRE}(a.P) \triangleq a.\mathrm{CRE}(P)$$

$$\mathrm{CRE}(\mathbf{go}_{\mathsf{A}} l.P) \triangleq l \qquad \mathrm{CRE}(P_1 \,|\, P_2) \triangleq \mathrm{CRE}(P_1) \odot \mathrm{CRE}(P_2)$$

$$\mathrm{CRE}(!P) \triangleq \mathrm{CRE}(P)^{\otimes}$$

Definition 3.2 (DFA Satisfaction) An agent P satisfies the DFA A, written $\vdash P : \mathsf{A}$, if $lang(\mathrm{CRE}(P)) \subseteq Acp(\mathsf{A})$ and, for every subagent of P of the form $\mathbf{go}_{\mathsf{A}'} l.Q$, it holds that $\vdash Q : \mathsf{A}'$. ■

In the full paper, we prove that the enforcement predicate can be established efficiently, while DFA satisfaction is decidable, but extremely hard to establish. This substantiate our hypothesis that verifying digests is preferable to inspecting the full code from the point of view computational complexity. We are now ready to state the soundness of this variation. It simply consists in finding a proper notion of well-formed systems. Like in Section 3.1, the entry policy can only express properties of single threads, instead of coalitions of threads hosted at a site. Thus, we modifiy rule (WF-G.SITE) from Figure 4 as below.

$$\frac{\begin{array}{l}(\text{WF-G.SITE}_{\mathrm{A}}) \\ \forall i \,.\, P_i \text{ a thread and } \quad \exists s \in S \,.\, lang(\mathrm{CRE}(P_i)) \subseteq Acp_s(M_p)\end{array}}{\vdash l[\![\, M \,\rangle\!\rangle\, P_1 | \ldots | P_n \,]\!] : \mathbf{ok}} \; l \text{ trustworthy}$$

This essentially requires that the languages associated to each of the threads in l are suffixes of words accepted by $\mathsf{M_p}$ (cf. Theorem 3.4 below). Since this may appear quite weak, it is worth remarking that the well-formedness predicate is just a 'consistency' check, a way to express that the agent is in a state from where it will respect the policy of l.

Theorem 3.3 (Subject Reduction for automata policies) *If* $\vdash N : \mathbf{ok}$ *and* $N \to N'$, *then* $\vdash N' : \mathbf{ok}$.

Theorem 3.4 (Safety for automata policies) *Let* N *be a well-formed system. Then, for every trustworthy site* $l[\![\, M \,\rangle\!\rangle\, P_1 | \ldots | P_n \,]\!]$ *in* N, *where each* P_i *is a thread, it holds that* $\sigma \in lang(CRE(P_i))$ *implies that there exists* $\sigma' \in Acp(\mathsf{M_p})$ *such that* $\sigma' = \sigma''\sigma$, *for some* σ''.

4 Resident Policies

Here we change the intended interpretation of policies. In the previous section a policy dictated the proposed behaviour of an agent *prior* to execution in a site, at the point of entry. This implied that safety in well-formed systems was a thread-wise property (see rules (WF-G.SITE$_{\mathrm{M}}$) and (WF-G.SITE$_{\mathrm{A}}$)). Here we focus on policies which are intended to describe the permitted (coalitional) behaviour of agents during execution at a site. Nevertheless these resident policies are still used to determine whether a new agent is allowed access to the site in question; entry will only be permitted if the addition of this incoming agent to the code currently executing at the site does not violate the policy.

Let us consider an example to illustrate the difference between entry and resident policies.

Example 4.1 Let LICENCE_SERV be the site name of a server that makes available K licences to download and install a software product. The distribution policy is based on a queue: the first K agents landing in the site are granted the licence, the following ones are denied. The policy of the server should be $\mathsf{M}_{\mathsf{p}}^{s} \triangleq \{\mathtt{get\_licence}^{K}\}$. However if this policy is interpreted as an entry policy, applying the theory of Section 3.1, then the system grants at most K licences to *each* incoming agent. Moreover this situation continues indefinitely, effectively handing out licences to all incoming agents. ■

We wish to re-interpret the policies of the previous section as *resident policies* and here we outline two different schemes for enforcing such policies. For simplicity we confine our attention to one kind of policy, that of multisets.

4.1 Static membranes

Our first scheme is conservative in the sense that many of the concepts developed in Section 3.1 for entry policies can be redeployed. Let us reconsider the migration rule from Figure 2:

$$\begin{array}{ll} \text{(R-MIG)} & k[\![\, M^k \,|\!\rangle\, \mathbf{go}_{\mathsf{T}} l.P \mid Q \,]\!] \;\parallel\; l[\![\, M^l \,|\!\rangle\, R \,]\!] \to \\ & \qquad k[\![\, M^k \,|\!\rangle\, Q \,]\!] \;\parallel\; l[\![\, M^l \,|\!\rangle\, P \mid R \,]\!] \qquad \text{if } \; M^l \vdash_{\mathsf{T}}^{k} P \end{array} \tag{2}$$

Here the membrane M^l only takes into consideration the incoming code P, and its digest T, when deciding on entry, via the predicate $M^l \vdash_{\mathsf{T}}^{k} P$. But if the membrane is to enforce a *resident policy*, then it must also take into account the contribution of the code already running in l, R. To do so we need a mechanism for *joining* policies, such as those of the incoming P and the resident R in (2). So let us assume that the set of policies, with the relation `enforces` is a partial order in which every pair of elements T_1 and T_2 has a *least upper bound*, denoted $\mathsf{T}_1 \sqcup \mathsf{T}_2$. For multiset policies this is the case as $\sqcup$ is simply multiset union. In addition we need to be

$$\text{(TI-EMPTY)}\ \Vdash \mathbf{nil} : \emptyset \qquad \text{(TI-ACT)}\ \frac{\Vdash P : \mathsf{T}}{\Vdash a.P : \mathsf{T} \cup \{a\}} \qquad \text{(TI-MIG)}\ \frac{\Vdash P : \mathsf{T}'}{\Vdash \mathbf{go}_{\mathsf{T}} l.P : \{l\}}\ \mathsf{T}' \ \texttt{enforces}\ \mathsf{T}$$

$$\text{(TI-REPL)}\ \frac{\Vdash P : \mathsf{T}}{\Vdash\ !P : \mathsf{T}^{\omega}} \qquad \text{(TI-PAR)}\ \frac{\Vdash P : \mathsf{T}_1 \qquad \Vdash Q : \mathsf{T}_2}{\Vdash P \mid Q : \mathsf{T}_1 \cup \mathsf{T}_2}$$

Fig. 7. Type inference for agents with policies as multisets

able to calculate the (minimal) policy which a process R satisfies; let us denote this as $\mathsf{pol}(R)$. For multiset policies we can adjust the rules in Figure 6, essentially by eliminating *weakening*, to perform this calculation; the resulting rules are given in Figure 7, with judgements of the form $\Vdash P : \mathsf{T}$.

Definition 4.1 Define the partial function $\mathsf{pol}(\cdot)$ over closed terms by letting $\mathsf{pol}(P)$ to be the unique policy such that $\Vdash P : \mathsf{T}$, if it exists. ■

With these extra concepts we can now change the rule (R-MIG) in (2) to take the current resident code into account. It is sufficient to change the side condition, from $M^l \vdash^k_{\mathsf{T}} P$ to $M^l, R \vdash^k_{\mathsf{T}} P$, where this latter is defined to be

$$\textbf{if } M^l_t(k) = \mathtt{good} \textbf{ then } (\mathsf{T} \sqcup \mathsf{pol}(R))\ \mathtt{enforces}\ \mathsf{M}^l_{\mathsf{p}} \textbf{ else } \vdash P \mid R : \mathsf{M}^l_{\mathsf{p}}$$

Here if only the digest needs to be checked then we compare $\mathsf{T} \sqcup \mathsf{pol}(R)$, that is the result of *adding* the digest to the policy of the resident code R, against the resident policy M_{p}. On the other hand if the source site is untrusted we then need to analyse the incoming code in parallel with the resident code R. It should be clear that the theory developed in Section 3.1 is readily adapted to this revised reduction semantics. In particular the Subject Reduction and Safety theorems remain true; we spare the reader the details. However it should also be clear that this approach to enforcing resident policies has serious practical drawbacks. An implementation would need to:

(i) freeze and retrieve the current content of the site, namely the agent R;

(ii) calculate the minimal policy satisfied by R to be merged with P's digest in order to check the predicate `enforces`, or typecheck the composed agent $P \mid R$;

(iii) reactivate R and, according to the result of the checking phase, activate P.

Even if the language were equipped with a *passification* operator, as in [17], the overall operation would still be computationally very intensive. Consequently we suggest below another approach.

4.2 Dynamic membranes

In the previous approach we have to repeatedly calculate the policy of the current resident code each time a new agent requests entry. Here we allow the policy in the membrane to "*decrease*," in order to reflect the resources already allocated to the resident code. So at any particular moment in time the policy currently in the membrane records what resources *remain*, for any future agents who may wish to enter; with the entry of each agent there is a corresponding decrease in the membrane's policy. Formally we need to change the migration rule (2) to one which not only checks incoming code, or digest, against the membrane's policy, but also updates the membrane:

$$\begin{array}{ll}(\text{R-MIG}') & k[\![\, M^k \,]\!\rangle\, \mathbf{go}_{\mathsf{T}} l.P \mid Q \,]\!] \;\parallel\; l[\![\, M^l \,]\!\rangle\, R \,]\!] \;\rightarrow \\ & \qquad\qquad k[\![\, M^k \,]\!\rangle\, Q \,]\!] \;\parallel\; l[\![\, \widehat{M^l} \,]\!\rangle\, P \mid R \,]\!] \quad \text{if} \;\; M^l \vdash^k_{\mathsf{T}} P \succ \widehat{M^l}\end{array}$$

where the judgement $M^l \vdash^k_{\mathsf{T}} P \succ \widehat{M^l}$ is defined as

$$\mathbf{let}\ \mathsf{T}' = \begin{cases} \mathsf{T} & \text{if } M^l_t(k) = \mathtt{good} \\ \mathsf{pol}(P) & \text{otherwise} \end{cases} \quad \mathbf{in}\ (\ \mathsf{T}'\ \mathtt{enforces}\ \mathsf{M}^l_{\mathsf{p}} \ \wedge \ \mathsf{M}^l_{\mathsf{p}} = \widehat{\mathsf{M}}^l_{\mathsf{p}} \sqcup \mathsf{T}' \ \wedge \ M^l_t = \widehat{M^l_t})$$

First notice that if this migration occurs then the membrane at the target site changes, from $\mathsf{M}^l_{\mathsf{p}}$ to $\widehat{\mathsf{M}}^l_{\mathsf{p}}$. The latter is obtained from the former by eliminating those resources allocated to the incoming code P. If the source site, k, is deemed to be $\mathtt{good}$ this is calculated via the incoming digest T; otherwise a direct analysis of the code P is required, to calculate $\mathsf{pol}(P)$.

This revised schema is more reasonable from an implementation point of view, but its soundness is more difficult to formalise and prove. As a computation proceeds no permanent record is kept in the system of the original resident policies at the individual sites. Therefore well-formedness can only be defined relative to an external record of what the resident policies were, when the system was initiated. For this purpose we use a function Θ, mapping trustworthy sites to policies; it is sufficient to record the original polices at these sites as we are not interested in the behaviour elsewhere.

Then we can define the notion of well-formed systems, relative to such a Θ; this is written as $\Theta \vdash N : \mathbf{ok}$ and the formal definition is given in Table 8. The crucial rule is (WF-G.SITE), for trustworthy sites. If l is such a site then $l[\![\, M \,]\!\rangle\, P \,]\!]$ is well-formed relative to the original record Θ if $\mathsf{M}^l_{\mathsf{p}} \sqcup \mathsf{pol}(P)$ guarantees the original resident policy at l, namely $\Theta(l)$.

Theorem 4.2 (Subject Reduction for resident policies) *If* $\Theta \vdash N : \mathbf{ok}$ *and* $N \rightarrow N'$, *then* $\Theta \vdash N' : \mathbf{ok}$.

(wf-g.site)
$$\frac{l \text{ trustworthy}}{\Theta \vdash l[\![\, M \,\rangle\!\rangle\, P \,]\!] : \mathbf{ok}\ (\mathsf{pol}(P) \sqcup \mathsf{M_p})\ \texttt{enforces}\ \Theta(l)}$$

(wf-empty)
$$\Theta \vdash \mathbf{0} : \mathbf{ok}$$

(wf-u.site)
$$\frac{}{\Theta \vdash l[\![\, M \,\rangle\!\rangle\, P \,]\!] : \mathbf{ok}}\ l \text{ not trustworthy}$$

(wf-par)
$$\frac{\Theta \vdash N_1 : \mathbf{ok}, \qquad \Theta \vdash N_2 : \mathbf{ok}}{\Theta \vdash N_1 \ \| \ N_2 : \mathbf{ok}}$$

Fig. 8. Well-formed systems under Θ

Theorem 4.3 (Safety for resident policies) *Let N be a well-formed system w.r.t. Θ. Then, for every trustworthy site $l[\![\, M \,\rangle\!\rangle\, P \,]\!]$ in N, $P \xrightarrow{\sigma} P'$ implies that* $\texttt{act}(\sigma)\ \texttt{enforces}\ \Theta(l)$.

5 Conclusion and Related Work

We have presented a framework to describe distributed computations of systems involving migrating agents. The activity of agents entering/running in 'good' sites is constrained by a membrane that implements the layer dedicated to the security of the site. We have described how membranes can enforce several interesting kind of policies. The basic theory presented for the simpler case has been refined and tuned throughout the paper to increase the expressiveness of the framework. Clearly, any other kind of behavioural specification of an agent can be considered a policy. For example, a promising direction could be considering logical frameworks (by exploiting model checking or proof checkers).

The calculus we have presented is very basilar: it is even simpler than CCS [12], as no synchronization can occur. Clearly, we did not aim at Turing-completeness, but at a very basic framework in which to focus on the rôle of membranes. We conjecture that, by suitably advancing the theory presented here, all the ideas presented here can be lifted to more complex calculi (including, e.g., synchronization, value passing and/or name restriction).

Related Work. In the last decade, several calculi for distributed systems with code mobility have appeared in literature. In particular, structuring a system as a (flat or hierarchical) collection of named sites introduced the possibility of dealing with sophisticated concrete features. For example, sites can be considered as the unity of *failure* [5,1], *mobility* [5,3] or *access control* [9,16,8]. The present work can be seen as a contribution to the last research line.

Similarly to [8], we have presented a scenario where membranes can evolve. However, the membranes presented in Section 4 only describe 'what is left' in the site. On the other hand, the (dynamically evolving) type of a site in [8] always constrains the overall behaviour of agents in the site and it is modified upon acquisition/loss of privileges through computations.

We borrowed from [16] the notion of *trust* between sites. In *loc. cit.*, agents coming from trusted sites are accepted without any control. Here, we relaxed this choice by examining the digest of agents coming from trusted sites. Moreover, we have a fixed net of trust; we believe that, once communication is added to our basic, the richer scenario of [16] (where the partial knowledge of a site can evolve during its computation) can be recovered.

A related paper is [11]. The authors develop a *generic* type system that can be smoothly instantiated to enforce several properties of the π–calculus (dealing with arity mismatch in communications, deadlock, race control and linearity). They work with one kind of types, and modify the subtyping relation in order to yield several relevant notions of safety. The main difference with our approach is that we have different kind of types (and, thus, different type checking mechanisms) for any variation we propose. It would be nice to lift our work to a more general framework closer to theirs; we leave this for future work.

Our work is also related to [15]. Policies are described there as deterministic finite automata and constrain the access to critical sections in a concurrent functional language. A type and effect system is provided that guarantees adherence of systems to the policy. In particular, the sequential behaviour of each thread is guaranteed to respect the policy, and the interleavings of the threads' locks to be safe. Differently from our paper, [15] has no code migration, and no explicit distribution; thus, only one centralised policy is used.

Membranes as filters between the computing body of a site and the external environment are also considered in [2,17]. There, membranes are computationally capable objects, and can be considered as a kind of processes. They can evolve and communicate both with the outer and with the inner part of the associated node, in order to regulate the life of the node. This differs from our conception of membranes as simple tools for the verification of incoming agents.

To conclude, we remark that our understanding of membranes is radically different from the concept of policies in [4]. Indeed, in *loc. cit.*, security automata control the execution of agents running in a site by *in-lined monitoring*. This technique consists in accepting incoming code unconditionally, but blocking at runtime those actions not abiding the site policy. Clearly, in order to implement the strategy, the execution of each action must be filtered by the policy. This contrasts with our approach, where membranes are 'containers' that regulate the interactions between sites and their environments. The computation taking place within the site is out of the control of the membrane that, hence, cannot rely on in-lined monitoring.

References

[1] R. Amadio. On modelling mobility. *Theoretical Computer Science*, 240(1):147–176, 2000.

[2] G. Boudol. A generic membrane model. Draft, 2004.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[4] U. Erlingsson and F. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proc. of New Security Paradigms Workshop*, pages 87–95. ACM, 1999.

[5] C. Fournet, G. Gonthier, J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. of CONCUR'96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.

[6] V. Garg and M. Raghunath. Concurrent regular expressions and their replationship to Petri nets. *Theoretical Computer Science*, 96:285–304, 1992.

[7] D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. Full version of this paper, available as Computer Science Research Report 02/2004, Dept. of Informatics, Univ. of Sussex (UK).

[8] D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *Proc. of ICALP'03*, volume 2719 of *LNCS*, pages 119–132. Springer-Verlag, 2003.

[9] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.

[10] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[11] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *Proceedings of POPL '01*, pages 128–141. ACM, 2001.

[12] R. Milner. *A Calculus for Communicating Systems*. Springer-Verlag, 1982.

[13] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[14] G. Necula. Proof-Carrying Code. In *Proc. of POPL '97*, pages 106–19. ACM, 1997.

[15] N. Nguyen and J. Rathke. A typed static analysis for a concurrent policy based resource access control. Draft, 2004.

[16] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL '99*, pages 93–104. ACM, 1999.

[17] A. Schmitt and J. Stefani. The M-calculus: a higher-order distributed process calculus. In *Proc. of POPL'03*, pages 50–61. ACM, 2003.