Static control of code migration

Francisco Martins[1] and António Ravara[2]

[1] Departamento de Matemática, Universidade dos Açores, Portugal.
fmartins@di.fc.ul.pt
[2] CLC and Departamento de Matemaática, Instituto Superior Técnico, Lisboa, Portugal.
aravara@ist.utl.pt

Abstract. This paper presents a type system to control the migration of code between sites in a concurrent distributed framework. The type system constitutes a decidable mechanism to ensure specific security policies, which control remote communication, process migration, and channel creation. The approach is as follows: each network administrator specifies sites privileges, and a type system checks that the processes running at those sites, as well as the composition of the sites, respect these policies. At runtime, well-typed networks do not violate the security policies declared for each site.

1 Introduction

The aim of this work is the development of a security mechanism to control the usage of system resources, such as memory and *cpu* cycles, in the presence of code mobility. We address this objective by means of a simple, decidable, and low complexity type system. The system should check the integrity and the consistency of user-declared security policies for networks, guaranteeing that well-typed networks are free of (runtime) security violation errors. Our approach consists in a simple set-based static analysis where the network administrators associate security policies to the sites they supervise, and by this mean, tailor the allowed interaction in a network.

Framework. To focus on the problem, and for simplicity sake, we develop our work using a mobile calculus tailored to the study of code migration in a distributed setting. A natural and simple framework to study distributed mobile systems is DPI, the distributed π-calculus of Hennessy and Riely [5], which extends the π-calculus [9, 10] by locating processes on a (flat) network of *sites*—named places where computation occurs. Communication only happens within sites (to avoid "global synchronisation"), but processes may migrate from one to another. However, DPI presents no notion of location of a resource: a global entity is responsible for allocating and managing memory.

A lexically scoped version of the distributed π-calculus—the $lsd\pi$ calculus [13]—addresses this drawback. It follows the DPI model in what respects the notions of locality (or *site*) and communication, but uses a lexically scoped regime for names that rules process migration. In this way, this calculus provides a *network-aware style of programming* in the sense that the time and the place associated with the creation of new resources (sites and channels) is explicit in the syntax of the programs. *What you see is what you get*, since the location of channels are clear in the program text, and the syntax plainly express the location of the resources used, even in presence of mobile code.

Therefore, migration of resources is a subset of the resource access operations, and thus is clearly bounded in the program's source code. Clients do not interact remotely with a server. Rather, they move to the site of the server and interact locally.

Thus, the $lsd\pi$ calculus seems suitable to study code migration, and to reason about resource usage, in a distributed setting. Code migration is triggered by channels themselves, rather than using explicit migration primitives (for an overview of distributed mobile calculi see a deliverable of an EU IST project [2]). We choose a monadic version of a flat calculus because the stress of our work is purely in the control of process migration rather than on communication, or on hierarchical issues.

Objectives. To control the migration of processes between sites, we use three security policies, to monitor, respectively, *remote communication*, *code migration*, and *channel creation*. These policies are directly related to the actions of the calculus, and therefore are independent from each other. *Remote communication* refers to the ability of a thread to send a message to a resource located at a distant site. *Code migration*, in turn, means that a thread may cross site boundaries, exiting its current site and entering a new one. This operation can be understood, from the source site point of view, as an upload of code. Finally, *channel creation* represents the ability of a thread to create a new channel in a foreign site. Mastering channel creation is important because (1) if a thread is able to create a channel on a remote site it means, as discussed later, that it would be able to migrate code to that destination, and (2) it may also give rise to a denial of service attack, if the source site creates an arbitrarily large number of channels in the destination site, consuming important resources, such as memory. Moreover, the topology of the network evolves dynamically, hence, sites may dynamically acquire new site acquaintances. Nonetheless, to these newly known localities a site may upload code, but it cannot download code from them—a 'write-down, no read up' policy.

The definition of a security policy for a given site consists on the enumeration of the sites allowed to perform the monitored actions. Thereafter, the type system checks whether these policies are followed by each process running in the site, and by the other known sites in the network that is being checked. This system is a tool to declare how code in those known sites may affect the computation of a given site, and to verify the compositionally of all these sites, constituting networks. The type system checks if the processes running at a given site respect its security policies, and if all the sites one wants to compose in a network will interoperate without violating each other policies.

2 The calculus

In a distributed setting one asks "to which locality does a channel belong to?". We adhere to the *lexical scope in a distributed context* of Obliq [3], which is here understood as the discipline under which the locality of channels is fixed throughout computation and can be determined by straightforward code inspection. This principle must not be disturbed by computation, in particular by code mobility. The paradigm allows us to *a priori* view channels as physical resources of localities, unlike in DPI where channels are network-wide identifiers that only *a posteriori* (via a type system) get located in sites.

$$
\begin{array}{rrcl}
\text{Channels} & & & a, b, x \in \mathcal{C} \\
\text{Sites} & & & r, s, t, y \in \mathcal{S} \\
\text{Values} & v & ::= & \mathit{val} \mid a@s \\
\text{Names} & n & ::= & a@s \mid s \\
\text{Sets of sites} & & & R, S, T \subseteq \mathcal{S} \\
\text{Site policies} & \mathcal{P} & ::= & \{\mathrm{rem} : R, \mathrm{mig} : S, \mathrm{new}\, T\} \\
\text{Processes} & P, Q & ::= & \mathsf{0} \mid a@s\,!\,\langle v\rangle \mid a@s?(v)\ P \mid a@s?*(v)\ P \mid \\
 & & & (P \mid Q) \mid (\boldsymbol{\nu} a@s)\ P \mid (\boldsymbol{\nu} s\colon \mathcal{P})\ P \\
\text{Networks} & N, M & ::= & \mathsf{0} \mid s\{\mathcal{P}\}\mathbf{[}P\mathbf{]} \mid (N \parallel M) \mid (\boldsymbol{\nu}_t n)\ N
\end{array}
$$

Fig. 1. Syntax of $lsd\pi$.

2.1 Syntax

The syntax of the calculus is described in Figure 1. Fix a denumerable set of *channels* $\mathcal{C}$ ranged over by a, b, x, and a denumerable set of *sites* $\mathcal{S}$ ranged over by r–t, y, disjoint from $\mathcal{C}$. A *channel* a located at a site s is denoted by $a@s$. The calculus presents two main syntactic categories: processes and networks. At process level we find the usual asynchronous π-calculus constructs [1, 7]; processes are built from the inactive process, $\mathsf{0}$, and from the asynchronous output process, $a@s\,!\,\langle v\rangle$, using three constructs: name restriction, $(\boldsymbol{\nu} a@s)\ P$ and $(\boldsymbol{\nu} s\colon \mathcal{P})\ P$, parallel composition, $(P \mid Q)$, and input, $a@s?(v)\ P$. We also include a form of replicated input, $a@s?*(v)\ P$.

Site policies $\mathcal{P}$ contain three kinds of policies: *remote communication*, *process migration*, and *name creation*. Each policy is related to an action of the calculus, and we proceed by enumerating the names of the sites that are allowed to perform these actions. Therefore, we relate remote communication, process migration, and name creation with the ability to output, input, and create channels, respectively.

Sites constitute the basic building blocks for networks. A network $s\{\mathcal{P}\}\mathbf{[}P\mathbf{]}$ denotes a site s, running a process P and protected by site policy $\mathcal{P}$. The set $\mathcal{P}$ defines the interactions allowed between s and the surrounding network. Networks are assembled using the *network parallel construct* $(N \parallel M)$. We use a different symbol from parallel processes (*c.f.* $(P \mid Q)$) to stress the fact that there is no communication between networks. The interaction between networks occurs through explicit migration of processes among sites. Name restriction $(\boldsymbol{\nu}_t n)\ N$ delimits name n, created at site t, to network N.

As an example, the following $lsd\pi$ term

$$r\{\mathcal{P}_1\}\mathbf{[}a@s\,!\,\langle b@r\rangle\mathbf{]} \parallel s\{\mathcal{P}_2\}\mathbf{[}a@s?(x@y)\ P\mathbf{]}$$

represents a network consisting of two sites r and s with security policies $\mathcal{P}_1$ and $\mathcal{P}_2$ (that we do not detail now). The output process running at site r is willing to deliver a message to the channel a from site s. As one may easily verify using the operational semantics we define ahead, the above network reduces in three steps to

$$r\{\mathcal{P}_1\}\mathbf{[}\mathsf{0}\mathbf{]} \parallel s\{\mathcal{P}_2\}\mathbf{[}P[b@r/x@y]\mathbf{]}$$

2.2 Semantics

The operational semantics of the calculus is presented following Milner *et al* [10]. We first define a *congruence relation* between networks that simplifies the *reduction relation* introduced thereafter.

Adding a distribution layer to a process calculus changes the structure of the resources (the channels). A resource is now (uniquely) associated to a locality, and this fact must be taken into account by the semantics of the calculus. DPI enforces the referred association via a system of dependent types. The syntax and the operational semantics do not clarify to which locality a resource belongs to, and furthermore, many processes that one may write are ruled out by the type system. The behavioural theory is also quite elaborate.

Our approach is based on a lexical scoping discipline that makes the location of resources clear in the program text, even in presence of mobile code. The extra structure of resources makes the definition of the semantics more intricate. We do not think one may keep the definition of free names of the π-calculus without taking into account that resources are distributed and located. Thus, instead of enforcing a correct use of names in processes via the type system only, we devise a semantics defined along the lines of the untyped λ-calculus, following Hindley and Seldin [6]. We believe this approach is simpler and provides clearer programs than those not based on lexical scoping.

Free names. We start by refining the definition of free names, taking into account the structure of located channels. Let w stand for both names and values (and consider an additional special value *nil*), and let X stand for both processes and networks. Considering that π ranges over $\{\mathrm{rem}, \mathrm{mig}, \mathrm{new}\}$, let $\mathsf{obj}(\{\pi_i : S_i \mid i \in I\}) \stackrel{\text{def}}{=} \bigcup_{i\in I} S_i$.

w	$\mathsf{subj}(w)$	$\mathsf{obj}(w)$	$\mathsf{chan}(w)$	$\mathsf{site}(w)$
s	$\mathcal{C}@s \cup \{s\}$	$\{s\}$	$-$	s
$a@s$	$\{a@s\}$	$\{a@s, s\}$	a	s
val	$\emptyset$	$\emptyset$	$-$	*nil*
nil	$\emptyset$	$\emptyset$	$-$	$-$

X	$\mathsf{fn}(X)$	$\mathsf{nm}(X)$
$\mathsf{0}$	$\emptyset$	$\emptyset$
$(X \mid Y)$	$\mathsf{fn}(X) \cup \mathsf{fn}(Y)$	$\mathsf{nm}(X) \cup \mathsf{nm}(Y)$
$(\boldsymbol{\nu}_t n\colon \mathcal{P})\ X$	$(\mathsf{fn}(X) \cup \{t\} \cup \mathsf{obj}(n) \cup \mathsf{obj}(\mathcal{P})) \setminus \mathsf{subj}(n)$	$\mathsf{nm}(X) \cup \{t\} \cup \mathsf{obj}(n) \cup \mathsf{obj}(\mathcal{P})$
$a@s?(v)\ P$	$\mathsf{fn}(P) \setminus \mathsf{subj}(\mathsf{site}(v)) \cup \mathsf{obj}(a@s)$	$\mathsf{nm}(P) \cup \mathsf{obj}(a@s) \cup \mathsf{obj}(v)$
$a@s\,!\,\langle v\rangle$	$\mathsf{obj}(a@s) \cup \mathsf{obj}(v)$	$\mathsf{obj}(a@s) \cup \mathsf{obj}(v)$
$s\{\mathcal{P}\}\mathbf{[}P\mathbf{]}$	$\mathsf{fn}(P) \cup \{s\} \cup \mathsf{obj}(\mathcal{P})$	$\mathsf{nm}(P) \cup \{s\} \cup \mathsf{obj}(\mathcal{P})$

Fig. 2. Free Names an Names in processes and in networks

Substitution. Equipped with a detailed notion of free names, the definition of substitution is a simple decision tree. We use the auxiliary notion of name binding: a name n binds another name m, $n{\hookrightarrow}m$, if $n \in \mathsf{obj}(m)$. Note that $n \in \mathsf{obj}(m)$ if, and only if, $m \in \mathsf{subj}(n)$.

However, to ensure that these concepts work, we consider that in the subprocess P of $a@s?(x@y)\ P$, one has $\mathsf{fn}(P) \cap \mathcal{C}@y = \{x@y\}$ and $\mathsf{fn}(P) \cap x@\mathcal{S} = \{x@y\}$. The forthcoming type system ensures these conditions.

$$(n)[m_2/m_1] \stackrel{\text{def}}{=} \begin{cases} m_2 & \text{if } n = m_1 \\ a@m_2 & \text{if } n = a@m_1 \\ n & \text{if } m_1 \not\rightarrow n \end{cases}$$

$$\mathsf{0}[m_2/m_1] \stackrel{\text{def}}{=} \mathsf{0}$$

$$(X \mid Y)[m_2/m_1] \stackrel{\text{def}}{=} (X[m_2/m_1] \mid Y[m_2/m_1])$$

$$((\boldsymbol{\nu}_t n\colon \mathcal{P})\ X)[m_2/m_1] \stackrel{\text{def}}{=} \begin{cases} (\boldsymbol{\nu}_t n\colon \mathcal{P})\ X & \text{if } (1) \\ (\boldsymbol{\nu}_{t[m_2/m_1]} n[m_2/m_1]\colon \mathcal{P}[m_2/m_1])\ X[m_2/m_1] & \text{if } \neg(1) \wedge (2) \\ (\boldsymbol{\nu}_{t[m_2/m_1]} n'\colon \mathcal{P}[m_2/m_1])\ X[n'/n][m_2/m_1] & \text{if } \neg(1) \wedge \neg(2) \wedge (3) \end{cases}$$

$$(a@s?(v)\ P)[m_2/m_1] \stackrel{\text{def}}{=} \begin{cases} a@s[m_2/m_1]?(v)\ P & \text{if } (4) \\ a@s[m_2/m_1]?(v)\ P[m_2/m_1] & \text{if } \neg(4) \wedge (5) \\ a@s[m_2/m_1]?(v')\ P[v'/v][m_2/m_1] & \text{if } \neg(4) \wedge \neg(5) \wedge (6) \end{cases}$$

$$(a@s\,!\,\langle v\rangle)[m_2/m_1] \stackrel{\text{def}}{=} a@s[m_2/m_1]\,!\,\langle v[m_2/m_1]\rangle$$

$$(s\{\mathcal{P}\}\mathbf{[}P\mathbf{]})[m_2/m_1] \stackrel{\text{def}}{=} s[m_2/m_1]\{\mathcal{P}[m_2/m_1]\}\mathbf{[}P[m_2/m_1]\mathbf{]}$$

1. $n \hookrightarrow m_1$ 2. $n \not\rightarrow m_2 \vee m_1 \notin \mathsf{fn}(X)$ 3. $n' \notin \{n, m, m_2\} \cup \mathsf{nm}(X)$
4. $v \hookrightarrow m_1$ 5. $v \not\rightarrow m_2 \vee m_1 \notin \mathsf{fn}(P)$ 6. $v' \notin \{m_1, m_2\} \cup \mathsf{nm}(a@s?(v)\ P)$

Fig. 3. Substitution on names, processes and networks

Structural congruence. The *structural congruence relation* is the least congruence relation closed under the rules in Figure 4. The rules are fairly standard. Networks are congruent up to α-renaming; the parallel composition operator for networks is taken to be commutative and associative, with 0 being the neutral element; scope may enlarge or reduce, provided that that are no capture of names. Notice that name extrusion between site and network level is only possible with the site that has created the name.

Reduction. The rules in figure 5 inductively define the *reduction relation* on $lsd\pi$ terms.

Rule RN-COMM is the *communication* rule of the calculus. It is defined only locally and it is the standard asynchronous π-calculus communication rule. Rule RN-COMR defines the communication for replicated inputs. Rules RN-MIGO, RN-MIGI, and RN-MIGR allow for processes to migrate across sites. When an input or an output operation is carried out over a remote resource, then the process migrates to the host site of the resource, since communication only arises locally.

The creation of a site, rule RN-NEWS, originate a new site only visible by its creator site. Rules RP-PAR and RN-RES allow for reduction to happen within network. Finally, rule RN-STR introduces structural congruence into the reduction relation.

The next example illustrates a download of code from a server site s requested by client site r. Assume that $r \notin \mathsf{fn}(Q)$.

$$r\{\mathcal{P}_1\}\mathbf{[}(\boldsymbol{\nu}\, req@r)\ dl@s\,!\,\langle req@r\rangle \mid req\,!\,\langle \mathit{val}\rangle\mathbf{]} \parallel s\{\mathcal{P}_2\}\mathbf{[}(dl@s?*(x@y)\ x@y?(\mathit{val})\ Q)\mathbf{]}$$

1. $N \equiv M$ if $N \equiv_\alpha M$
2. $(N \parallel M) \parallel M' \equiv N \parallel (M \parallel M'), \quad M \parallel N \equiv N \parallel M, \quad N \parallel \mathsf{0} \equiv N$
3. $(\boldsymbol{\nu}_t n)\ N \parallel M \equiv (\boldsymbol{\nu}_t n)\ (N \parallel M)$ if $n \notin \mathsf{fn}(()M)$
   $(\boldsymbol{\nu}_s b@r)\ s\{\mathcal{P}\}[P] \equiv s\{\mathcal{P}\}[(\boldsymbol{\nu} b@r)\ P]$
   $(\boldsymbol{\nu}_t a@s)\ (\boldsymbol{\nu}_u b@r)\ N \equiv (\boldsymbol{\nu}_u b@r)\ (\boldsymbol{\nu}_t a@s)\ N$
   $(\boldsymbol{\nu}_t s)\ (\boldsymbol{\nu}_u a@r)\ N \equiv (\boldsymbol{\nu}_u a@r)\ (\boldsymbol{\nu}_t s)\ N$ if $s \neq r$ and $t \neq u$
   $(\boldsymbol{\nu}_t s)\ (\boldsymbol{\nu}_u r)\ N \equiv (\boldsymbol{\nu}_u r)\ (\boldsymbol{\nu}_t s)\ N$ if $s \neq u$ and $t \neq r$
4. $s\{\mathcal{P}\}[P \mid Q] \equiv s\{\mathcal{P}\}[P] \parallel s\{\mathcal{P}\}[Q]$

Fig. 4. Structural congruence on networks.

$$s\{\mathcal{P}\}[a?(v_1)\ P \mid a\,!\,\langle v_2\rangle] \to s\{\mathcal{P}\}[P[v_2/v_1]] \qquad \text{(RN-COMM)}$$

$$s\{\mathcal{P}\}[a?*(v_1)\ P \mid a\,!\,\langle v_2\rangle] \to s\{\mathcal{P}\}[a?*(v_1)\ P \mid P[v_2/v_1]] \qquad \text{(RN-COMR)}$$

$$s\{\mathcal{P}_1\}[\mathsf{0}] \parallel r\{\mathcal{P}_2\}[a@s\,!\,\langle v\rangle] \to s\{\mathcal{P}_1\}[a@s\,!\,\langle v\rangle] \parallel r\{\mathcal{P}_2\}[\mathsf{0}], \quad r \neq s \qquad \text{(RN-MIGO)}$$

$$s\{\mathcal{P}_1\}[\mathsf{0}] \parallel r\{\mathcal{P}_2\}[a@s?(v)\ Q] \to s\{\mathcal{P}_1\}[a@s?(v)\ Q] \parallel r\{\mathcal{P}_2\}[\mathsf{0}], \quad r \neq s \qquad \text{(RN-MIGI)}$$

$$s\{\mathcal{P}_1\}[\mathsf{0}] \parallel r\{\mathcal{P}_2\}[a@s?*(v)\ Q] \to s\{\mathcal{P}_1\}[a@s?*(v)\ Q] \parallel r\{\mathcal{P}_2\}[\mathsf{0}], \quad r \neq s \qquad \text{(RN-MIGR)}$$

$$s\{\mathcal{P}_1\}[(\boldsymbol{\nu} r\colon \mathcal{P}_2)\ P] \to (\boldsymbol{\nu}_s r)\ (s\{\mathcal{P}_1\}[P] \parallel r\{\mathcal{P}_2\}[\mathsf{0}]) \qquad \text{(RN-NEWS)}$$

$$\frac{N \to M}{N \parallel M' \to M \parallel M} \qquad \frac{N \to M}{(\boldsymbol{\nu}_t n)\ N \to (\boldsymbol{\nu}_t n)\ M} \qquad \text{(RN-PAR,RN-RES)}$$

$$\frac{N \equiv N' \quad N' \to M' \quad M' \equiv M}{N \to M} \qquad \text{(RN-STR)}$$

Fig. 5. Reduction rules.

The client issues a new request to the dl (download) resource of the server by communicating a fresh channel req. The server upon the received request migrates process Q to the server using the acquired channel. Finally, the client fires the downloaded process. Reduction is as follows. Security annotations were deliberately omitted since they play no role in reduction.

$$r[(\boldsymbol{\nu}\, req@r)\ dl@s\,!\,\langle req@r\rangle \mid req@r\,!\,\langle \mathit{val}\rangle] \parallel s[dl@s?*(x@y)\ x@y?(\mathit{val})\ Q] \to$$
(RN-STR, RN-MIGO)

$$(\boldsymbol{\nu}_r req@r)\ r[req@r\,!\,\langle \mathit{val}\rangle] \parallel s[dl@s?*(x@y)\ x@y?(\mathit{val})\ Q \mid dl@s\,!\,\langle req@r\rangle] \to$$
(RP-COMR)

$$(\boldsymbol{\nu}_r req@r)\ r[req@r\,!\,\langle \mathit{val}\rangle] \parallel s[dl@s?*(x@y)\ x@y?(\mathit{val})\ Q \mid req@r?(\mathit{val})\ Q] \to$$
(RN-MIGI)

$$(\boldsymbol{\nu}_r req@r)\ r[req@r\,!\,\langle \mathit{val}\rangle \mid req@r?(\mathit{val})\ Q] \parallel s[dl@r?*(x@y)\ x@y?(\mathit{val})\ Q] \to$$
(RP-COMM, RN-STR)

$$r[(\boldsymbol{\nu}\, req@r)\ Q] \parallel s[dl@s?*(x@y)\ x@y?(\mathit{val})\ Q]$$

3 Type system

The type system we present enforces the user-defined security policies in $lsd\pi$ networks. We guarantee that, at runtime, well-typed networks do not violate the specified security policies.

Examples. In the following, we present some examples of erroneous networks that should be rejected by the type system. Consider, in all examples, that sites denoted by r, s, and t represent distinct locations.

A remote communication error occurs whenever an output to a located channel is performed from a site not belonging to the **rem** policy of the remote site. The next example elucidates this situation.

Example 1. Consider network $s\{\mathrm{rem} : \{t\}\}$**[**$\mathbf{0}$**]** $\| \; r\{\mathcal{P}\}$**[**$a@s \,!\, \langle x@y \rangle$**]**. The output process running at site r is willing to send a remote message to site s; however, this action is not allowed, since r is not mentioned in the **rem** policy of s. The inclusion of r in the policies of s fix the problem: $s\{\mathrm{rem} : \{t, r\}\}$**[**$\mathbf{0}$**]** $\| \; r\{\mathcal{P}\}$**[**$a@s \,!\, \langle x@y \rangle$**]**.

Code migration is controlled using policy keyword **mig** and specifying which sites are allowed to upload code.

Example 2. The network $s\{\mathrm{mig} : \{t\}\}$**[**$P$**]** $\| \; r\{\mathcal{P}\}$**[**$a@s?(x@y) \; \mathbf{0}$**]** is rejected because site s denies migration of code from site r as process $a@s?(x@y) \; \mathbf{0}$ intents. Including r in the policies of s overcomes the problem: $s\{\mathrm{mig} : \{t, r\}\}$**[**$P$**]** $\| \; r\{\mathcal{P}\}$**[**$a@s?(x@y) \; \mathbf{0}$**]**.

The creation of remote channels is controlled by the policy keyword **new**, enumerating the sites authorised to create remote channels.

Example 3. Network $s\{\mathrm{new} : \{t\}\}$**[**$\mathbf{0}$**]** $\| \; r\{\mathcal{P}\}$**[**$(\boldsymbol{\nu} \; a@s) \; a@s \,!\, \langle b@s \rangle$**]** fails to type check because site s denies creation of remote channels (as well as remote communications) from site r. Network $s\{\mathrm{new} : \{t, r\}, \mathrm{rem} : \{r\}\}$**[**$\mathbf{0}$**]** $\| \; r\{\mathcal{P}\}$**[**$(\boldsymbol{\nu} \; a@s) \; a@s \,!\, \langle b@r \rangle$**]** is well typed.

A more trickier situation resulting from code migration is shown below.

Example 4. Consider network $s\{\mathcal{P}\}$**[**$b@r?(x@y) \; a@s \,!\, \langle x@y \rangle$**]** $\| \; r\{\mathrm{mig} : \{s\}\}$**[**$\mathbf{0}$**]**. The remote message $a@s \,!\, \langle x@y \rangle$ is going to run at site r, since it is the continuation of a process that migrates from s to r. Although r grants migration privileges to s, it does not allow for remote communications from s, and therefore the network should be rejected. Fix the security fault including $\mathrm{rem} : \{s\}$ into the policies for site r.

Runtime errors. In what follows we formalise the notion of runtime error.

Definition 1 (runtime errors). *Let* $r \neq s$, $\mathcal{E} = \{N | N \rightarrow^{\star} \nu u_1 \ldots \nu u_k (M' \| M)\}$, *and* M *of the form*

$$r\{\mathcal{P}_1\}\textbf{[}P\textbf{]} \| \; s\{\mathcal{P}_2\}\textbf{[}a@r \,!\, \langle v \rangle\textbf{]}, \;\; s \notin \mathcal{P}_1(\mathrm{rem}) \tag{1}$$

$$r\{\mathcal{P}_1\}\textbf{[}P\textbf{]} \| \; s\{\mathcal{P}_2\}\textbf{[}a@r?(x@y) \; P\textbf{]}, \;\; s \notin \mathcal{P}_1(\mathrm{mig}) \tag{2}$$

$$r\{\mathcal{P}_1\}\textbf{[}P\textbf{]} \| \; s\{\mathcal{P}_2\}\textbf{[}(\boldsymbol{\nu} \; a@r) \; P\textbf{]}, \;\; s \notin \mathcal{P}_1(\mathrm{new}) \tag{3}$$

$$\varphi ::= \{a_i : \gamma_i\}_{i \in I} \quad \text{site types} \qquad V ::= \gamma@S \mid \mathrm{val}\,@\emptyset \quad \text{value types}$$
$$\gamma ::= \mathrm{ch}(V)^t \quad \text{channel types} \qquad t ::= o \mid i \mid b \quad \text{channel tags}$$

Fig. 6. Syntax of types.

$$\mathrm{val}\,@\emptyset \leq \mathrm{val}\,@\emptyset \qquad \frac{\gamma_1 \leq \gamma_2 \quad S_1 \subseteq S_2}{\gamma_1@S_1 \leq \gamma_2@S_2} \qquad \text{(\textit{Value subtyping})}$$

$$\frac{k = i, b \quad V_1 \leq V_2}{\mathrm{ch}(V_1)^k \leq \mathrm{ch}(V_2)^i} \qquad \frac{k = o, b \quad V_2 \leq V_1}{\mathrm{ch}(V_1)^k \leq \mathrm{ch}(V_2)^o} \qquad \frac{V_1 \leq V_2 \quad V_2 \leq V_1}{\mathrm{ch}(V_1)^b \leq \mathrm{ch}(V_2)^b}$$

(*Channel subtyping*)

Fig. 7. Subtyping relation.

Networks that violate a security policy are in the set $\mathcal{E}$. Our type safety result (theorem 3) guarantees that well-typed do not belong to $\mathcal{E}$.

Types. The syntax for types is depicted in figure 6. We record types for sites. A *site type* φ is a mapping from the free channels of the site to channel types. A *channel type* γ traces the type of the values that are communicated along the channel, as well as its usage (input, output, or both); Channels may carry other channels or basic values, as described by *value types* V. The *type for channel values* assumes the form $\gamma@S$, where γ is the type of the channels that can be carried, and S is the set of sites hosting the communicated channels. *Channel tags* i, o, and b denote, respectively, that a channel is used for input, for output, or for both input and output purposes.

Subtyping. The binary relation $\leq$ on types is defined following Pierce and Sangiorgi [12] as the least preorder relation closed under the rules in figure 7. The original intuitions remain unchanged, namely that the subtyping relation is covariant for inputs, contravariant for outputs, and invariant if the channels are used both for input and for output purposes.

Typing. Figures 8 and 9 present the typing rules for processes and networks.

A *typing* Γ is a partial function of finite domain from site names to pairs $(\varphi, \mathcal{P})$ of site types and site policies. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ. The typing $\Gamma + x\colon T$ denotes the type environment Γ' such that $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{x\}$, $\Gamma'(y) = \Gamma(y)$ for $y \neq x$, and $\Gamma'(x) = T$ when $x \notin \mathrm{dom}(\Gamma)$, otherwise $\Gamma'(x) = \Gamma(x) + T$. The addition of site types, $\gamma_1 + \gamma_2$, is simple the disjoit set union of γ_1 and γ_2.

In process $a@s?(x@y)\ P$ name y is called an *alias*. In fact, y do not denote a site, but it is a placeholder the for sites communicated along $a@s$. When verifying security policies the alias y makes no sense. The policies are not written in term of aliases (bound names) but in term of (real) sites. Therefore, aliases must be resolved to the sites they stand for. We keep a cross reference between aliases and site names in set Δ. A site stands for itself, *i.e.*, $\Delta(s) = \{s\}$, otherwise, $\Delta(s)$ is the set of sites that may instatiate alias s during reduction.

We make use of some auxiliary notation to compress the writing of the typing rules. The first and second projections of pair $\Gamma(s) = (\varphi, \mathcal{P})$ is denoted as $\Gamma(s)_1$ and $\Gamma(s)_2$, respectively. We write $\Gamma(a@s)$ for $\Gamma(s)_1(a)$. To decompose a located channel $a@s$, one

$$\textsc{SP-Nil}\ \frac{}{\Gamma;\Delta \vdash_s \mathbf{0}} \qquad \textsc{SP-Par}\ \frac{\Gamma;\Delta \vdash_s P \qquad \Gamma;\Delta \vdash_s Q}{\Gamma;\Delta \vdash_s P \mid Q}$$

$$\textsc{SP-Out}\ \frac{\begin{array}{c}\Gamma(a@r) \leq \mathrm{ch}(\Gamma(v)@\Delta(\mathsf{site}(v)))^o \\ \Delta(s) \subseteq \Gamma(r)_2(\mathrm{rem})\end{array}}{\Gamma;\Delta \vdash_s a@r\,!\,\langle v\rangle} \qquad \textsc{SP-Inpr}\ \frac{\Gamma;\Delta \vdash_s a@r?(v)\ P}{\Gamma;\Delta \vdash_s a@r?*(v)\ P}$$

$$\textsc{SP-Inp}\ \frac{\begin{array}{c}S \stackrel{\mathrm{def}}{=} \Delta(\mathsf{site}(()v)) \qquad \mathcal{P} \stackrel{\mathrm{def}}{=} \{k\colon \bigcap_{t\in S} \Gamma(t)_2(k) \mid k \in \{\mathrm{rem}, \mathrm{mig}, \mathrm{new}\}\} \\ \Gamma(a@r) \leq \mathrm{ch}(\gamma@S)^i \qquad \Delta(s) \subseteq \Gamma(r)_2(\mathrm{mig}) \\ \Gamma + \mathsf{site}(v)\colon (\{\mathsf{chan}(v)\colon \gamma\}, \mathcal{P}); \Delta + \mathsf{site}(v)\colon S \vdash_r P\end{array}}{\Gamma;\Delta \vdash_s a@r?(v)\ P}$$

$$\textsc{SP-Resc}\ \frac{\Gamma + r\colon (\varphi + a\colon \gamma, \mathcal{P}); \Delta \vdash_s P \qquad \Delta(s) \subseteq \mathcal{P}(\mathrm{new})}{\Gamma + r\colon (\varphi, \mathcal{P}); \Delta \vdash_s (\boldsymbol{\nu} a@r)\ P}$$

$$\textsc{SP-Ress}\ \frac{\Gamma + r\colon (\emptyset, \mathcal{P}); \Delta + r\colon \{r\} \vdash_s P}{\Gamma;\Delta \vdash_s (\boldsymbol{\nu} r\colon \mathcal{P})\ P}$$

Fig. 8. Typing processes.

uses $\mathsf{site}(a@s) = s$ and $\mathsf{chan}(a@s) = a$. The $\mathsf{site}(\mathit{val}) = \mathsf{chan}(\mathit{val}) = undef$. Finally, we assume that $\Delta(undef) = \emptyset$ and that adding $undef$ elements to a typing or to a site type produce no effects.

The type system includes two kinds of judgements: (a) judgement $\Gamma;\Delta \vdash_s P$ means that process P is running at site s and is well typed under typing assumptions Γ, resolving aliases as specified in Δ; and (b) judgement $\Gamma \vdash N$ denotes that network N is well typed under typing assumptions Γ.

An output process $a@r\,!\,\langle v\rangle$ is well-typed if the type of channel a, located at r, has at least output capabilities and is enough (a subtype) to carry value v. In the case that v is a located channel, then a must be able to carry values located at that (group of) site(s). Moreover, site r must allow any site that s stands for to remote communicate with it. Notice that we never use the name of a site to check security polices, as we do not distint syntactically sites from aliases. So, we always resolve the name using set Δ.

To type an input process $a@r?(x@y)\ P$, rule SP-Inp, we type P considering it is running in r, since $a@r$ triggers the migration of P from s to r. Furthermore, typing Γ is augmented with the site of value v. The type of $\mathsf{site}(v)$ deserves to be discussed. First of all, if v is *val*, then neither Γ nor Δ are affected. So, letting v be $b@t$, we add site t to Γ, announcing b as its unique known channel. This restriction prevents the usage of channels located in t without being explicitly created. As for the type policy, we compute the set of policies that every site that may instanciate t agrees. This way, we guarantee that P will not violate the security policies in any circumstances. Finally, we add a link between alias t and the sites that may communication through $a@r$. On what concerns the checking of security, we monitor that the migration operation to r is allowed from every site denoted by s. The following network type-checks,

$$s\{\emptyset\}[a@r?(x@y)\ x@y\,!\,\langle c@s\rangle] \parallel r\{\mathrm{mig}\colon \{s\}\}[a@r\,!\,\langle b@t\rangle] \parallel t\{\mathrm{rem}\colon \{r\}\}[\mathbf{0}]$$

$$\text{SN-NIL}\ \frac{}{\Gamma \vdash 0} \qquad \text{SN-PAR}\ \frac{\Gamma \vdash N \quad \Gamma \vdash M}{\Gamma \vdash N \parallel M}$$

$$\text{SN-NET}\ \frac{\Gamma; \{s\colon \{s\}\} \vdash_s P \quad \Gamma(s) = (\varphi, \mathcal{P})}{\Gamma \vdash s\{\mathcal{P}\}[P]} \qquad \text{SN-RESS}\ \frac{\Gamma + r\colon (\emptyset, \mathcal{P}) \vdash N}{\Gamma \vdash (\boldsymbol{\nu}_s r\colon \mathcal{P})\ N}$$

$$\text{SN-RESC}\ \frac{\Gamma + r\colon (\varphi + a\colon \gamma, \mathcal{P}) \vdash N \quad s \in \mathcal{P}(\text{new})}{\Gamma + r\colon (\varphi, \mathcal{P}) \vdash (\boldsymbol{\nu}_s a@r)\ N}$$

Fig. 9. Typing networks.

The typing rules for the inaction process, the parallel process, and the replicated input are fairly standard. The creation of channels requires the authorisation from the site where the channel is being created. Notice that we do not use directly the name of the site, but resolve it through set Δ.

The typing rules for networks are found in figure 9. Rule SN-NET types a located process in a site $s\{\mathcal{P}\}[P]$. Process P must be well typed under type assumptions Γ, where the processes is running in s, having a link established to itself. Moreover, we demand that the network policies defined at network level match exactly the policies formulated in Γ—no process is allowed to forge security policies. When restricting a channel, SN-RESC, we check that the destination site allows the operation. Notice the role of the subscript site of the new construct. The remaining rules are straightforward.

Results on the type system. The results we present are standard and the proofs proceed directly by induction.

The type and subtype system rules are syntax oriented, so an algorithm to compute types (in polynomial time) can be found just by reading the rules backward. Notice that there are no recursive types.

Theorem 1 (Decidability of the type system). *Given Γ and N, the problem of verifying whether $\Gamma \vdash N$ is decidable.*

Reduction preserves the typability of processes and of networks. This result uses a standard substitution lemma, together with subject congruence.

Lemma 1. *If $\Gamma \vdash N$ and $N \equiv M$, then $\Gamma \vdash M$.*

Proof. We proceed by induction on the type derivation, analysing the last rule applied.

Theorem 2 (Subject reduction). *If $\Gamma \vdash N$ and $N \to M$, then $\Gamma \vdash M$.*

Proof. By induction on the inference of $N \to M$. We proceed by case analysis on the reduction relation and examine the last typing rule of the typing derivation. The proof is straightforward.

The following result states that well-typed networks do not belong to $\mathcal{E}$.

Theorem 3 (type safety). *If $\Gamma \vdash N$ and $N \to^\star M$, then $M \notin \mathcal{E}$.*

Proof. The proof is direct and proceeds by absurd.

4 Conclusions

Summary. We present a type system to control resource access, in particular via the migration of code, in a distributed mobile calculus. We monitor three security policies: remote communication, process migration, and channel creation, corresponding to the actions of the calculus, these policies enabling us to control code migration. The security policies are defined by the site administrators, following an intuitive and easy approach. For each site, its administrator specifies what operations other sites are allowed to perform.

The current setting allow us to focus on the security policies for resources. Using $lsd\pi$, we present a non-trivial, yet simple and low-cost, solution based on typing and subtyping relations. The system checks that processes running at given sites respect their security policies and that sites in a network interoperate correctly, without violating each other policies. Specifically, we prove subject reduction, define runtime errors, and then prove type safety.

Further work. This work is a further step towards static control of code migration. It smoothly extends (and simplifies) our previous work, which dealt only with a predefined number of sites and which did not allow passing site identities [8].

We plan further developments along two main directions: (a) the definition of security policies at resource level and therefore be able to refine the interaction between sites; (b) and the ability to adjust security policies dynamically.

Related work. Other approaches to resource security in distributed mobile calculi comprise DPI and KLAIM [4, 11]. See [2] for a general survey on concurrent mobile calculus, type systems, and security policies. DPI possesses an explicit objective construct to move code—the **go** primitive. The control of migration is found along three aspects: a keyword **mig**, a subtype relation, and the ability to communicate site names. If a process "sees" the **mig** keyword as part of the type of a site, then it may migrate code to that site. The subtype relation, together with the capability to communicate site names, allows a site to tailor the information (*e.g.* resource names, control keywords) that the target site would be able to use. From a programming point of view, this approach does not seem very attractive since security annotations are spread along the code and it is difficult to understand what actions are really allowed to execute. It is not clear how to implement type inference.

KLAIM uses a capability type system to control operations on tuple spaces. Each site defines the actions that other sites can perform. There is a correspondence between the capabilities and the calculus's actions. For the migration primitive (*eval*) the type specifies also the security restrictions that the migrating process should obey. The KLAIM approach is similar to ours in the sense that security policies are declared at site level, but differs substantially when we consider the way policies are programmed and checked. Notice that the KLAIM type system is far more complex than ours is, although it provides roughly the same guarantees. One main distinction concerns the place where the security policies are defined: security policies in KLAIM talk about what operations a site may perform on other sites, whereas in our framework each site talks about what actions it allows others to perform on it. From the site administrator point of view this looks more appropriate.

Moreover, our system is tailored to the particular aspects of lexical scoped settings.

Acknowledgements. This work was partially supported by the EU FEDER and the Portuguese Fundação para a Ciência e a Tecnologia (via CLC and the project Space-TimeTypes, POCTI/EIA/55582/2004), and the EU IST proactive initiative FET-Global Computing (projects Mikado, IST–2001–32222, and Profundis, IST–2001–33100).

References

1. G. Boudol. Asynchrony and the π-calculus. Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.
2. G. Boudol, I. Castellani, F. Germain, and M. Lacoste. Models of distribution and mobility: State of the art. Mikado Deliverable D1.1.1, 2002.
3. L. Cardelli. A language with distributed scope. In *POPL'95: 22nd Annual ACM Symposium on Principles of Programming Languages (San Francisco, CA, U.S.A.)*, pages 286–297. ACM Press, 1995.
4. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *Proc. of 30th ICALP'03*, volume 2719 of *Lecture Notes in Computer Science*, pages 119–132. Springer-Verlag, 2003.
5. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.
6. J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.
7. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.
8. F. Martins and A. Ravara. Typing migration control in $lsd\pi$. In *Proceedings of FCS'04*, pages 1–12, Turku, Finland, 2004.
9. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. Springer-Verlag, 1993.
10. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992.
11. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 1(240):215–254, 2000.
12. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
13. A. Ravara, A. Matos, V. Vasconcelos, and L. Lopes. Lexically scoping distribution: what you see is what you get. In *FGC'03*, volume 85(1) of *ENTCS*, July 2003.