Typing migration control in $lsd\pi$.

Francisco Martins[1] and António Ravara[2]

[1] Department of Mathematics, University of Azores, Portugal.
[2] CLC and Department of Mathematics, Instituto Superior Técnico, Lisboa, Portugal.

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 its sites privileges, and the type system checks that both processes and the composition of sites respect these policies. At runtime, well-typed networks do not violate the security policies declared for each site.

1 Introduction

A natural and simple framework to study distributed mobile systems is Dpi, the distributed π-calculus of Hennessy and Riely [4], which extends the π-calculus [6, 7] by distributing processes over a (flat) network of *localities*—named places where computation occurs. Communication only happens within localities (to avoid "global synchronization"), 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. DiTyCO, a distributed extension of the TyCO programming language [5, 11], 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. The main motivations of DiTyCO are: (1) a *network-aware style of programming* in the sense that the time and place associated with the creation of new resources (sites and channels) is explicit in the syntax of the programs; and (2) an easier implementation, based on current technology by providing the compiler with information to generate code for the creation and the access to these resources. 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 eliminating the costs of maintaining long remote sessions between clients and servers. This mechanism can be further optimised by dispatching light-weight triggers that collect/transmit information by order of centralised processes. However, DiTyCO lacks a security mechanism to control the usage of important resources such as memory and *cpu* cycles. This work addresses such lacuna, but to keep focus on the problem, we turn our attention to a distributed version of the pi-calculus—the $lsd\pi$ calculus [10]—that is the theoretical basis of DiTyCO. This calculus seems more suitable to study code migration in a distributed setting than other proposals [1], since this migration is triggered by channels themselves, rather than using an explicit **go** primitive.

A natural way to control code mobility is by means of a type system. Our main motivation is to devise a simple, decidable, and low complexity type system to check the integrity

and consistency of user-declared security policies, guaranteeing that well-typed networks are free of (runtime) security violation errors.

The security policies we envisage protect the actions of the calculus from unauthorised usage of site resources. We monitor the ability to send and receive messages and to create remote channels. 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 between sites in a network. The definition of a security policy consists on the enumeration, for each site, of the sites allowed to perform the monitored actions. Thereafter, the type system checks whether these policies are followed by each thread.

To control the migration of processes between sites we propose three security policies: *remote communications*, *code migrations*, and *channel creation*. *Remote communication* denotes the ability of a thread to send a message to a resource located at a distant site. *Code migration*, by its turn, means that a thread may cross site boundaries, exiting its current site and entering a new site. This operation can be understood, from the source site point of view, as an upload of code. *Channel creation* represents the ability to create channels in a foreign site. Mastering this action is important because (1) if a thread is able to create a channel on a remote site it means, as discussed later, that it is able to migrate code to that destination, and (2) it may also give rise to a denial of service attack, where the source site creates an arbitrarily large number of channels in the destination site, consuming important resources, such as memory.

In the lexically scoped distributed pi-calculus, $lsd\pi$ [10], the computational space is partitioned into a flat structure of named sites hosting concurrent processes (or threads). Thus, sites arise as shells of computations where processes are executed. Moreover, sites form also communicational units for the exchange of messages—communication is local to sites—and security units, in the sense that processes located at a given site share common security policies. Remote communications must be explicitly programmed: the interaction between threads located at different sites occurs through migration of code and messages across that sites. In $lsd\pi$ channels are resources associated uniquely to sites, located at creation time. A resource may be referred locally or remotely. Local resources are identified by simple identifiers; remote resources are mentioned using located identifiers—channel-site pairs—that indicate explicitly the sites where the resources are placed. Although one can use a located identifier to indicate a local resource, the motto is that simple identifiers refer local resources, whereas located identifiers are associated to remote resources. Herein, we use a sub-calculus of $lsd\pi$ that fits the current version of DiTyCO.

Other approaches to resource security in distributed mobile calculi comprise Dpi [3, 4] and Klaim [2, 8]. See [1] for a general survey on concurrent mobile calculus, type systems, and security policies. Dpi possesses an explicit objective construct to migrate code—the **go** primitive. The control of migration is found along tree 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 for a site to tailor the information (*e.g.* resource names, control keywords) that the target site is able to

use. From a programming point of view, this approach does not seems very attractive since security annotations are spread along the code and it is difficult to understand what actions are really allowed to perform. The type system appears to be quite difficult to implement.

Klaim uses a capability type system to control operations on tuple spaces. For each site it is defined the allowed actions that can be performed in other sites. There is a correspondence between the capabilities and the calculus 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. One main distinction concerns the place where the security policies are defined: security policies in Klaim talk about what operations are allowed for a site to perform on other sites, whereas in our framework each site talks about what actions it allows for others to perform on it. From the site administrator point of view this looks more adequate.

2 Syntax

This section describes the calculus as well as its types. We present their syntax and some examples of networks, hinting informally the semantics of $lsd\pi$.

2.1 The calculus

The $lsd\pi$ calculus extends the *pi*-calculus [7], distributing processes over flat networks of named sites where they compute. Communications occur by message passing through channels, that we call resources, but only within a site. Resources maintain a fixed location throughout the computation and are located at creation time. We choose a flat and monadic calculus because the stress of our work is purely in the control of process migration rather than on communication or on hierarchical issues. Indeed, the selection of DiTyCO lead us to a clear understanding of the security issues underlying code migration and let us settle the basis for reasoning about resource usage.

Unique to the calculus is the notion of lexical scoping, a well-known feature from main-stream programming languages like, for instance, Pascal, C, Java, or ML. In $lsd\pi$ this means that we may refer a channel without site annotation whenever we are at its home site. Therefore, a channel may be addressed by its simple name—the channel name—or by its located (or global) name—the pair channel name–host site—whenever the reference is made from a foreign site.

The syntax of $lsd\pi$ is described in figure 1. Fix a denumerable set of *simple channels*, $\mathcal{C}$, ranged over by a, b, c, x, y, z, and a denumerable set of *sites*, $\mathcal{S}$, range over by r, s, t, disjoint from $\mathcal{C}$. *Channels*, may be simple—a—or located—$a@s$—, meaning a channel a from a site s. *Processes* are the standard *pi*-calculus processes, apart from the input process, $u?(x : S)\ P$, that mentions a set of sites S. This set plays an important role in the assurance of security policies: it restricts the channels that may instantiate x, enumerating the sites allowed to host them. Notice, however, that in $lsd\pi$ sites are not first class citizens, in the sense that

$$
\begin{array}{rrll}
\text{simple channels} & & & a, b, c, x, y, z \in \mathcal{C} \\
\text{sites} & & & r, s, t \in \mathcal{S} \\
\text{channels} & u & ::= & a \mid a@s \\
\text{values} & v & ::= & () \mid u \\
\text{globals} & g, h & ::= & a@s \mid s \\
\text{sets of sites} & & & S, R, T \subseteq \mathcal{S} \\
\text{processes} & P, Q & ::= & \mathsf{0} \mid u\,!\,\langle v \rangle \mid u?(x : S)\ P \mid u?*(x : S)\ P \mid \\
 & & & P \,|\, Q \mid (\boldsymbol{\nu}\, u)\ P \\
\text{networks} & N, M & ::= & \mathsf{0} \mid s_G[P] \mid N \parallel M \mid (\boldsymbol{\nu}\, a@s)\ N
\end{array}
$$

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

they are not passed around. The claim is that there is no purpose to reveal a site location. Instead, disclosing a port at a site (a located channel) is all that is needed to establish a link to it.

The basic building blocks of networks are sites running processes. A network $s_G[P]$ means a site s running a process P using the security policies bound by G. The set G defines the interactions allowed between s and the surrounding network. For more about G, see the section on types. Networks are put together using the *network parallel construct* $N \parallel M$. We use a different symbol from parallel processes (*c.f.* $P \,|\, Q$) to stress the fact that there is no communication between networks. The interaction between networks occurs through explicit migration of processes among sites. The other constructs allow us to restrict located channels in a network, $(\boldsymbol{\nu}\, a@s)\ N$, and to talk about an inactive network, $\mathsf{0}$.

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

$$r_{G_1}[a@s\,!\,\langle b \rangle] \parallel s_{G_2}[a?(x : S)\ P]$$

This term represents a network consisting of two sites r and s with security policies G_1 and G_2, respectively. The process running at r is willing to deliver a message at channel a from s. So, in order to perform that, this process must migrate first from r to s and then communicate with a within s. Notice that in $a@s\,!\,\langle b \rangle$ one gets a clear understanding of the sites each name belongs to: the located channel $a@s$ is hosted by s, and simple channel b is from r. One important thing to remember is that simple channels are always channels of the site that hosts the process where they are mentioned. With no surprise, the above network reduces to

$$r_{G_1}[\mathsf{0}] \parallel s_{G_2}[P[b@r/x]]$$

We skip the details for now, but notice that r is explicitly mentioned in the reference to channel b, since it "left home".

2.2 The types

The syntax for types is given in figure 2. A *typing* is a mapping from site names to pairs of site types and site policies. Our approach is to specify security policies at site level, possibly

$$
\begin{array}{rcl}
\text{typings } \Gamma & ::= & \{s_1 : (\varphi_1, G_1), \ldots, s_n : (\varphi_n, G_n)\} \\
\text{site types } \varphi & ::= & \{a_1 : \gamma_1, \ldots, a_n : \gamma_n\} \\
\text{site policies } G & ::= & \{\text{rem} : S_1, \text{mig} : S_2, \text{new} : S_3\} \\
\text{channel types } \gamma & ::= & \text{ch}(\gamma)@S^t \mid \text{val} \\
\text{site tags } t & ::= & o \mid i \mid b
\end{array}
$$

Fig. 2. Syntax of types.

by the site security administrator, that set up a kind of "border control" between the site and the neighbouring network. We consider three sorts 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.

Site types are mappings from channel, the free names of the site, to channel types. A channel type records the type of the argument and the sites where this channel may be used. We need this information to be able to check security policies. For instance, to type an output process, say $x\,!\,\langle v \rangle$, running at site s, we require that the sites of the channels that may instantiate x allow s to remote communicate with them.

The *tags*, i for *input*, o for *output*, and b for *both*, are part of a subtype relation (inspired on proposal of Pierce and Sangiorgi [9]) on the set of sites that may instantiate a given channel (defined in section 4). For example, this subtype relation says that is safe to use a channel of type $\text{ch}(\gamma)@\{s, r\}^o$ whenever it is possible to use a channel of type $\text{ch}(\gamma)@\{s\}^o$.

3 Semantics

The operational semantics of the calculus is presented following Milner *et al* [7]. We first define a *congruence relation* between processes and networks that simplifies the *reduction relation* introduced thereafter.

Free and bound names, as well as the substitution relation that lies upon them, present some subtleties, introduced by located identifiers and by lexical scoping, which deserve some attention. The conceptual ideas behind bindings are the following.

– At network level, the binding of a located channel, entails the binding of free occurrences of both located channel anywhere in the network, and also of simple channel at its host site.

$$(\boldsymbol{\nu}\, a@r)\ \Big(s[\ldots a \ldots a@r \ldots] \ \parallel\ r[\ldots a \ldots a@r \ldots]\Big)$$

– At site level, the binding of a channel (simple or located) only binds the free occurrences (simple or located, respectively) of this channel.

$$s\Big[(\boldsymbol{\nu}\, a)\ (\ldots a \ldots a@s \ldots)\Big] \qquad\qquad s\Big[(\boldsymbol{\nu}\, a@s)\ (\ldots a \ldots a@s \ldots)\Big]$$

1. $N \equiv M$ if $N \equiv_\alpha M$
2. $((N \parallel M) \parallel M') \equiv (N \parallel (M \parallel M'))$
3. $(M \parallel N) \equiv (N \parallel M)$
4. $(N \parallel \mathsf{0}) \equiv N$
5. $((\boldsymbol{\nu}\, a@s)\ N) \parallel M \equiv (\boldsymbol{\nu}\, a@s)\ (N \parallel M)$ if $a@s \notin \mathrm{fn}(M)$
6. $(\boldsymbol{\nu}\, a@r)\ (\boldsymbol{\nu}\, b@s)\ N \equiv (\boldsymbol{\nu}\, b@s)\ (\boldsymbol{\nu}\, a@r)\ N$
7. $(\boldsymbol{\nu}\, a@r)\ s_G[P] \equiv s_G[(\boldsymbol{\nu}\, a@r)\ P]$ if $(r \neq s \wedge s \in G_1(\mathrm{new})) \vee (r = s \wedge a \notin \mathrm{fn}(P))$
8. $(\boldsymbol{\nu}\, a@s)\ s_G[P] \equiv s_G[(\boldsymbol{\nu}\, a)\ P]$ if $a@s \in \mathrm{fn}(P)$
9. $s_G[a@s\,!\,\langle v\rangle] \equiv s_G[a\,!\,\langle v\rangle]$
10. $s_G[a@s?(x:S)\ P] \equiv s_G[a?(x:S)\ P]$
11. $s_G[a@s?*(x:S)\ P] \equiv s_G[a?*(x:S)\ P]$

Fig. 3. Structural congruence on networks.

For complete definitions and comments on free and bound names in $lsd\pi$ refer to [10].

3.1 Structural congruence

Definition 1 (Structural congruence). *The* structural congruence relation *is the least congruence relation closed under the rules in figures 3 and 4.*

The first six rules in figure 3 are fairly standard. Networks are congruent up to α-renaming; the parallel composition operator for networks is taken to be commutative and associative, with neutral element $\mathsf{0}$.

Scope extrusion rules, however, deserve a more detailed analysis. In rule 7, if the located channel belongs to the site where it is going to move in or out, then there must not exists a simple channel with the same name in P. A similar concern is expressed in rule 8. The reason for these side conditions become clear in the following examples. These two networks should not be in the congruence relation

$$(\boldsymbol{\nu}\, a@s)\ s_G[a\,!\,\langle b\rangle \mid a@s\,!\,\langle c\rangle] \ \not\equiv\ s_G[(\boldsymbol{\nu}\, a@s)\ a\,!\,\langle b\rangle \mid a@s\,!\,\langle c\rangle]$$

since the (left-hand side) binder, at network level, binds both the simple and the located channel, whereas the binder at process level only binds the located channel.

Similarly,

$$(\boldsymbol{\nu}\, a@s)\ s_G[a\,!\,\langle b\rangle \mid a@s\,!\,\langle c\rangle] \ \not\equiv\ s_G[(\boldsymbol{\nu}\, a)\ a\,!\,\langle b\rangle \mid a@s\,!\,\langle c\rangle]$$

1. $P \equiv Q$ if $P \equiv_\alpha Q$

2. $((P \mid Q) \mid R) \equiv (P \mid (Q \mid R))$

3. $(P \mid Q) \equiv (Q \mid P)$

4. $(P \mid \mathsf{0}) \equiv P$

5. $((\boldsymbol{\nu}\, u)\ P) \mid Q \equiv (\boldsymbol{\nu}\, u)\ (P \mid Q)$ if $u \notin \mathrm{fn}(Q)$

6. $(\boldsymbol{\nu}\, u)\ (\boldsymbol{\nu}\, u')\ P \equiv (\boldsymbol{\nu}\, u')\ (\boldsymbol{\nu}\, u)\ P$

7. $(\boldsymbol{\nu}\, a)\ \mathsf{0} \equiv \mathsf{0}$

Fig. 4. Structural congruence on processes.

the (right-hand side) binder, at site level, only binds the simple channel.

The last three rules rename located channels to simple channels (and vice-versa) when the channels are mentioned from their home site.

In the $lsd\pi$ calculus there is also a rule for splitting and regrouping sites: $s_G[P] \parallel s_G[Q] \equiv s_G[P \mid Q]$. However, in this version of the calculus, where sites are constant, that rule is covered by the migration rules RN-Migi, RN-Migo, and RN-Migr ahead (see figure 5).

Structural congruence on processes (figure 4) presents a similar set of rules when compared to structural congruence on networks. The only remark concerns rule 7 where "garbage collection" is only allowed locally. One could think that a more general rule also stands, say $(\boldsymbol{\nu}\, u)\ \mathsf{0}$, but this is not the case, since $\mathrm{fn}((\boldsymbol{\nu}\, u)\ \mathsf{0}) \neq \mathrm{fn}(\mathsf{0})$ when u is a located channel. Recall that s is free in $(\boldsymbol{\nu}\, a@s)\ \mathsf{0}$. This reasoning applies also to networks, where garbage collection would also have the malicious effect of erasing site policy annotations.

3.2 Reduction

We use contexts for processes and networks to simplify the reduction relation.

Definition 2 (Reduction contexts).

$$\begin{array}{rcl} E & ::= & [\cdot] \mid (E \mid P) \mid (\boldsymbol{\nu}\, u)\ P \\ F & ::= & [\cdot] \mid (F \parallel N) \mid (\boldsymbol{\nu}\, a@s)\ P \end{array}$$

Definition 3 (Translation of identifiers). *The translation of free names from site r to site s, σ_{rs}, is a total function defined as follows.*

$$\begin{array}{l} \sigma_{rs}(t) = t \qquad \sigma_{rs}(a@s) = a \\ \sigma_{rs}(a) = a@r\ \sigma_{rs}(a@t) = a@t, \quad t \notin \{r, s\} \end{array}$$

Definition 4 (Reduction relation). *The rules in figure 5 inductively define the reduction relation on $lsd\pi$ terms.*

$$\text{RP-Comm} \quad a?(x:T)\ P \mid a\,!\,\langle v\rangle \to P[v/x]$$

$$\text{RP-Comr} \quad a?*(x:T)\ P \mid a\,!\,\langle v\rangle \to a?*(x:T)\ P \mid P[v/x]$$

$$\text{RN-Migo} \quad s_{G_1}[P] \parallel r_{G_2}[a@s\,!\,\langle v\rangle \mid Q] \to s_{G_1}[P \mid (a@s\,!\,\langle v\rangle)\sigma_{rs}] \parallel r_{G_2}[Q], \qquad r \neq s$$

$$\text{RN-Migi} \quad s_{G_1}[P] \parallel r_{G_1}[(a@s?(x:A)\ Q) \mid R] \to s_{G_1}[P \mid (a@s?(x:A)\ Q)\sigma_{rs}] \parallel r_{G_2}[R], \quad r \neq s$$

$$\text{RN-Migr} \quad s_{G_1}[P] \parallel r_{G_1}[(a@s?*(x:A)\ Q) \mid R] \to s_{G_1}[P \mid (a@s?*(x:A)\ Q)\sigma_{rs}] \parallel r_{G_2}[R],\ r \neq s$$

$$\text{RP-Cont} \quad \frac{P \to Q}{E[P] \to E[Q]}$$

$$\text{RP-Str} \quad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

$$\text{RN-Site} \quad \frac{P \to Q}{s_G[P] \to s_G[Q]}$$

$$\text{RN-Cont} \quad \frac{N \to M}{F[N] \to F[M]}$$

$$\text{RN-Str} \quad \frac{N \equiv N' \qquad N' \to M' \qquad M' \equiv M}{N \to M}$$

Fig. 5. Reduction rules.

RP-Comm is the *communication* rule of the calculus. It is defined only locally and it is the standard asynchronous *pi*-calculus communication rule. The rules RN-Migi, RN-Migo, and RN-Migr allow for processes to migrate across sites. When an input or output operation is carried out over a remote resource then, since communication only arises locally, the process migrates to the host site of the resource. In order to keep lexical scoping, we apply function, σ, to translate the free names of the migrating process. If we are moving from site r to site s, then σ_{rs} transforms the references to channels from r into located channels, since they will be mentioned from s, and makes references to channels from s into simple channels.

Rules RP-Str and RN-Str introduce structural congruence into the reduction relation and are crucially used to bring processes and networks into the form requested by the left-hand-side of axioms RP-Comm, RN-Migo, RN-Migi, and RN-Migr. Finally, rules RP-Cont, RN-Site, and RN-Cont allow for reduction to happen within process and network contexts.

The next example illustrates a code download from a server site *srv* requested by client site *cl*. Assume that $req \notin \mathrm{fn}(Q)$.

$$cl[(\boldsymbol{\nu}\, newreq)\ download@srv\,!\,\langle newreq\rangle \mid newreq\,!\,\langle\rangle \mid P] \parallel srv[(download?*(req)\ req?()\ Q) \mid R]$$

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

$$\begin{aligned}
cl[(\boldsymbol{\nu}\, newreq)\ download@srv\,!\,\langle newreq\rangle \mid newreq\,!\,\langle\rangle \mid P] \parallel \\
srv[download?*(req)\ req?()\ Q \mid R] \quad \rightarrow \\
(\boldsymbol{\nu}\, newreq@cl)\ cl[newreq\,!\,\langle\rangle \mid P] \parallel \\
srv[download?*(req)\ req?()\ Q \mid R \mid download\,!\,\langle newreq@cl\rangle] \quad \rightarrow \\
(\boldsymbol{\nu}\, newreq@cl)\ cl[newreq\,!\,\langle\rangle \mid P] \parallel \\
srv[download?*(req)\ req?()\ Q \mid newreq@cl?()\ Q \mid R] \quad \rightarrow \\
(\boldsymbol{\nu}\, newreq@cl)\ cl[newreq\,!\,\langle\rangle \mid P \mid newreq?()\ Q\sigma_{srv,cl}] \parallel \\
srv[download?*(req)\ req?()\ Q \mid R] \quad \rightarrow \\
cl[(\boldsymbol{\nu}\, newreq)\ Q\sigma_{srv,cl} \mid P] \parallel \\
srv[download?*(req)\ req?()\ Q \mid R] \quad \rightarrow
\end{aligned}$$

4 Type system

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

4.1 Examples

In the following, we present some examples of erroneous networks that should be caught by the type system's sieve. 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 four examples elucidate this sort of situation.

Example 1.

$$s_{\{\text{rem}:\{t\}\}}[P] \parallel r_{G_1}[a@s\,!\,\langle x\rangle]$$

This example illustrates the common remote communication error. 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.

Example 2.

$$s_{\{\text{rem}:\{t\}\}}[b@r?(x : S)\ a@s\,!\,\langle x\rangle] \parallel r_{\{\text{mig}:\{s\}\}}[\mathsf{0}]$$

The above example shows a more tricky situation. Notice that the remote message $a@s\,!\,\langle x\rangle$ should be understood as running at site r, since it is prefixed by an input migrates from s to r. Although site r grants migrations from s to r, site s does not allow for remote communications from r, and therefore this network should be rejected.

Example 3.

$$s_{\{\text{rem}:\{r\}\}}[a?(x : \{t\})\ \mathsf{0}] \parallel r_{G_1}[a@s\,!\,\langle b\rangle]$$

Another kind of communication errors arise when a process is trying to pass on information about a non-reliable site. Considering a mailing system as an example, we can decide to grant a certain machine to deliver messages, but restrict the sites from where these messages come from.

The example above is illegal, since the output process, $a@s\,!\,\langle b\rangle$, at site r, is communicating information about a channel from site r, but the input process running at site s only admits information that mentions channels from t (indicated by $x : \{t\}$).

Example 4.

$$s_{\{\text{rem}:\{r\}\}}[a?(x : \{r, t\})\ x\,!\,\langle c\rangle] \parallel r_{\emptyset}[a@s\,!\,\langle b\rangle]$$

We analyse the output process, $x\,!\,\langle c\rangle$, at site s. Variable x may be instantiated with channels from r or channels from t. Then, to type-check the network above correctly, we need to be able to remote communicate with sites r and t from site s. But it fails because site r does not concede any remote communication privilege.

Likewise remote communication, the control of code migration (carried out through an input process) is performed using a (different) policy keyword—**mig**—and specifying which sites are allowed to upload code. The next example depicts this case.

Example 5.

$$s_{\{\text{mig}:\{t\}\}}[P] \parallel r_{G_1}[a@s?(x : S)\ Q]$$

The above network should be rejected because site s denies migration of code from site r as it is intended by process $a@s?(x : S)\ Q$.

The creation of remote channels is controlled using the policy keyword **new** and enumerating the sites authorised to create remote channels. For instance, the following network fails to type check because site s denies creation of remote channels from site r.

Example 6.

$$s_{\{\text{new}:\{t\}\}}[P] \parallel r_{G_1}[(\boldsymbol{\nu}\ a@s)\ Q]$$

$$b \leq i \qquad b \leq o$$

$$\frac{R \subseteq S}{S^o \leq R^o} \qquad \frac{S \subseteq R}{S^i \leq R^i} \qquad \frac{t \leq t'}{S^t \leq S^{t'}}$$

$$\gamma \leq \gamma \qquad \frac{\gamma_1 \leq \gamma_2 \quad \gamma_2 \leq \gamma_3}{\gamma_1 \leq \gamma_3}$$

$$\frac{\gamma_1 \leq \gamma_2 \quad S^t \leq R^{t'}}{\mathrm{ch}(\gamma_1)@S^t \leq \mathrm{ch}(\gamma_2)@R^{t'}}$$

Fig. 6. Subtyping relation.

$$\text{SS-Chl} \quad \Gamma \vdash_s a@r : \Gamma(r)_1(a)$$

$$\text{SS-Chs} \quad \Gamma \vdash_s a : \Gamma(s)_1(a)$$

Fig. 7. Typing channels.

4.2 Subtyping

The binary relation $\leq$ on types is defined following Pierce and Sangiorgi [9] and is the least equivalence relation closed under the rules in figure 6. Intuitively, this subtype relation allows for the inclusion of more site identifiers (of where a channel can be located in) when we perform outputs, and allows for the exclusion of some site identifiers when we perform inputs. If a channel is used both for input and output its type is fixed.

Tags i, o, and b mark the usage of channels and denote, respectively, that a channel is used for input, for output, or for both input and output purposes. The relation is defined conventionally: covariant for inputs, contravariant for outputs, and invariant when a channel is used for inputs and outputs.

4.3 Typing

Figures 7, 8, and 9 present, respectively, the typing rules for channels, processes, and networks.

We record types for sites.[3] Therefore, the types of the free channels are kept within the types of the sites where they belong to. Types for located channels are directly fetched from their identifiers; to access types for simple channels we need extra information identifying their host site. We keep track of this information in the typing judgements for channels. In fact, the judgement $\Gamma \vdash_s v : \gamma$, means that if v is a simple channel, its host site is s. Of course, if v is located, it already has all the information needed for typing. Look up in figure 7 for the typing rules for channels.

Judgements for processes, $\Gamma \vdash_{s,S} P$, besides the identity of the current site s, remember also the set of sites where P might be hosted at runtime. This is a crucial information for checking security policies because S give us the sites where events (remote communication,

[3] A word on notation: let $\Gamma(s)_1$ and $\Gamma(s)_2$ be the first and second projections of the pair $\Gamma(s) = (\varphi, G)$.

$$\textsc{SP-Outl}\quad \frac{\Gamma(r)_1(a) = \mathrm{ch}(\gamma_1)@\{r\}^b \qquad \Gamma \vdash_s v : \gamma_2 \qquad \gamma_2 \leq \gamma_1 \qquad S \subseteq \Gamma(r)_2(\mathrm{rem})}{\Gamma \vdash_{s,S} a@r\,!\,\langle v\rangle}$$

$$\textsc{SP-Outs}\quad \frac{\Gamma(s)_1(a) = \mathrm{ch}(\gamma_1)@R^b \qquad \Gamma \vdash_s v : \gamma_2 \qquad \gamma_2 \leq \gamma_1 \qquad S \subseteq \Gamma(r)_2(\mathrm{rem}),\ \forall r \in R}{\Gamma \vdash_{s,S} a\,!\,\langle v\rangle}$$

$$\textsc{SP-Inpl}\quad \frac{\Gamma(r)_1(a) = \mathrm{ch}(\gamma_1)@\{r\}^b \qquad \Gamma(s)_1(x) = \mathrm{ch}(\gamma_2)@T^b \qquad \Gamma \vdash_{s,\{r\}} P \qquad \mathrm{ch}(\gamma_2)@T^b \leq \gamma_1 \qquad S \subseteq \Gamma(r)_2(\mathrm{mig})}{\Gamma \setminus x@s \vdash_{s,S} a@r?(x : T)\ P}$$

$$\textsc{SP-Inps}\quad \frac{\Gamma(s)_1(a) = \mathrm{ch}(\gamma_1)@R^b \qquad \Gamma(s)_1(x) = \mathrm{ch}(\gamma_2)@T^b \qquad \Gamma \vdash_{s,R} P \qquad \mathrm{ch}(\gamma_2)@T^b \leq \gamma_1 \qquad S \subseteq \Gamma(r)_2(\mathrm{mig}),\ \forall r \in R}{\Gamma \setminus x@s \vdash_{s,S} a?(x : T)\ P}$$

$$\textsc{SP-Nil}\quad \Gamma \vdash_{r,S} \mathsf{0}$$

$$\textsc{SP-Par}\quad \frac{\Gamma \vdash_{s,S} P \qquad \Gamma \vdash_{s,S} Q}{\Gamma \vdash_{s,S} (P \,|\, Q)}$$

$$\textsc{SP-Ress}\quad \frac{\Gamma(s)_1(a) = \mathrm{ch}(\gamma)@S^b \qquad \Gamma \vdash_{s,S} P}{\Gamma \setminus a@s \vdash_{s,S} (\boldsymbol{\nu}\, a)\ P}$$

$$\textsc{SP-Resl}\quad \frac{\Gamma(r)_1(a) = \mathrm{ch}(\gamma)@\{r\}^b \qquad \Gamma \vdash_{s,S} P \qquad S \setminus s \subseteq \Gamma(r)_2(\mathrm{new})}{\Gamma \setminus a@r \vdash_{s,S} (\boldsymbol{\nu}\, a@r)\ P}$$

Fig. 8. Typing processes.

code migration, or channel creation) take place. We proceed by explaining the typing rules for processes that can be found in figure 8.

The output process, $a@r\,!\,\langle v\rangle$, is well-typed if: 1) the type of a, located at r, is a channel type having as arguments a supertype of the type of v, and if site r allows any site where the output process might be located at runtime to remote communicate with it. Following is an example of an instance of the output rule

$$\frac{\begin{array}{c}\Gamma \vdash_s v : \mathrm{ch}(\gamma)@\{s\}^b \\ \mathrm{ch}(\gamma)@\{s\}^b \leq \mathrm{ch}(\gamma)@\{s,r\}^i \\ \Gamma(r)_1(a) = \mathrm{ch}(\mathrm{ch}(\gamma)@\{s,r\}^i)@\{r\}^b \\ \{t\} \subseteq \{s,t\}\end{array}}{\Gamma \vdash_{s,\{t\}} a@r\,!\,\langle v\rangle}$$

On the other hand, if the output is performed over a a simple channel, say a, we require every site where a may be located to give permission for remote communications from the sites where the process, $a\,!\,\langle v\rangle$, may be at runtime. It sounds as a contradiction that a channel may be hosted by more than one site, but that is not the case, the point is that it may not be possible to determine the site that hosts a channel at compile time. Consider a communication over a channel received as a parameter.

$$\text{SN-Nil} \qquad \Gamma \vdash \mathsf{0}$$

$$\text{SN-Par} \qquad \frac{\Gamma \vdash N \qquad \Gamma \vdash M}{\Gamma \vdash (N \parallel M)}$$

$$\text{SN-Net} \qquad \frac{\Gamma \vdash_{s,\{s\}} P \qquad \Gamma(s)_2 = G \qquad \Gamma(s)_1(a) = \mathrm{ch}(\gamma)@\{s\}^b,\ \forall a \in \mathrm{dom}(\Gamma(s)_1)}{\Gamma \vdash s_G[P]}$$

$$\text{SN-Resl} \qquad \frac{\Gamma \vdash N \qquad S \setminus r \subseteq \Gamma(r)_2(\mathrm{new}) \qquad S \text{ is the set of sites where } a@r \text{ occurs free in } N}{\Gamma \setminus a@r \vdash (\boldsymbol{\nu}\, a@r)\ N}$$

Fig. 9. Typing networks.

To type an input process, $\Gamma \vdash_{s,S} a@r?(x : T)\ P$, we type P resolving simple channels to site s and considering P as running in r, since $a@r$ triggers the migration of P from s to r. Bear in mind that simple channels remain bound to s by lexical scoping. We check that the migration operation to r is allowed from every site where the process may be located at runtime. The following network type-checks,

$$s_\emptyset[a@r?(x : \{t\})\ x\,!\,\langle c\rangle] \parallel r_{\{\mathrm{mig}:\{s\}\}}[a\,!\,\langle b@t\rangle] \parallel t_{\{\mathrm{rem}:\{r\}\}}[\mathsf{0}]$$

The typing rules of the inaction process, the parallel process, and the creation of a local channels are fairly standard. The creation of remote channels requires the authorisation from the site where the channel is being created. This authorisation must be issued to all the sites in S.

The typing rules for networks can be found in figure 9. Rule SN-Net types a located process in a site, $s_G[P]$. Process P must be well typed under type assumptions Γ, where simple channels are considered resources from site s and the processes is running in s. 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. The remaining precondition assures that visible resources of the site, addressed only by simple channels, are indeed located at the site.

The handling of resource restrictions at network level is somehow more delicate than at site level, since we lack information about the site that indeed create it. Therefore, we require that the site hosting the resource must concede creation permissions to every site that uses the resource. Notice in the following example that since $a@s$ is free in site r, site s must grant remote creation privileges to r.

$$(\boldsymbol{\nu}\, a@s)\ s_{\{\mathrm{new}:\{r\},\mathrm{rem}:\{r\}\}}[\mathsf{0}] \parallel r_\emptyset[a@s\,!\,\langle b\rangle]$$

where $S = \{r\}$, $\Gamma(s) = \{(\emptyset, \{\mathrm{new} : \{r\}, \mathrm{rem} : \{r\}\})\}$, and $\Gamma(r) = \{(b : \mathrm{ch}(\gamma)@\{r\}^b, \emptyset)\}$.

Reduction preserves the typability of processes and networks.

Lemma 1. *If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.*

Proof. We proceed by induction on the type derivation, analysing the last rule applied. We just sketch case 7. (figure 3, page 6), where the last typing rule applied is SP-Resl, to illustrate the use of the side condition $s \in G_1(\text{new})$ when $a@s \notin \text{fn}(P)$. Consider the case where $(\boldsymbol{\nu}\, a@r)\ s_G[P] \equiv s_G[(\boldsymbol{\nu}\, a@r)\ P]$.

By hypothesis, $\Gamma \vdash (\boldsymbol{\nu}\, a@r)\ s_G[P]$. Assuming $s \neq r$, we need to consider to subcases:

- $a@r \in \text{fn}(P)$
  Therefore, by rule SP-Resl, $s \in \Gamma(r)_2(\text{new})$ and so, by SP-Resl and SN-Net, we prove that $\Gamma \vdash s_G[(\boldsymbol{\nu}\, a@r)\ P]$.
- $a@r \notin \text{fn}(P)$
  In this case $s \notin S$, but the side condition $s \in G_1(\text{new})$ ensures that r gives permission to create a local channel from s and so we can apply SP-Resl and SN-Net safely.

The case when $s = r$ does not matter because we may always allow to create a channel in the site that hosts it. Notice that we exclude the current site when checking the sites that must grant the new policy.

Theorem 1 (Subject reduction).

1. *If $\Gamma \vdash_{s,S} P$ and $P \to Q$, then $\Gamma \vdash_{s,S} Q$.*
2. *If $\Gamma \vdash N$ and $N \to M$, then $\Gamma \vdash M$.*

Proof. By induction on the typing derivation of $\Gamma \vdash_{s,S} P$ and of $\Gamma \vdash N$. We proceed by case analysis on the reduction relation and examine the last typing rule of the typing derivation. The proof is straightforward.

Theorem 2 (Decidability of the type system). *The type system presented in figures 7, 8, and 9 is decidable.*

Proof. 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.

4.4 Runtime errors

Our type system guarantee that well-typed networks do not violate the specified security policies. In what follows we formalise the notion of runtime error.

Definition 5 (runtime errors). *Assume that $r \neq s$. Let*

$$\mathcal{E} = \{N | N \to^\star \nu \boldsymbol{X}(M' \parallel M)\}$$

and M of the form

$$r_{G_1}[\![P]\!] \parallel s_{G_2}[\![a@r\,!\,\langle v\rangle]\!],\ s \notin G_1(\text{rem}) \tag{1}$$

$$r_{G_1}[\![P]\!] \parallel s_{G_2}[\![a@r?(x:T)\ P]\!],\ s \notin G_1(\text{mig}) \tag{2}$$

$$s_G[\![(a?(x:T)\ P) \mid a\,!\,\langle b@r\rangle]\!],\ r \notin T \tag{3}$$

$$r_{G_1}[\![P]\!] \parallel s_{G_2}[\![(\boldsymbol{\nu}\, a@r)\ P]\!],\ s \notin G_1(\text{new}) \tag{4}$$

The set $\mathcal{E}$ contains the networks that violate at least one security policy. The following result states that well-typed networks do not belong to $\mathcal{E}$.

Corollary 1 (type safety). *If $\Gamma \vdash N$, and $N \rightarrow^{\star} M$, then $M \notin \mathcal{E}$.*

Proof. The proof is straightforward and proceeds by absurd.

5 Conclusions and future work

The type system that we present allows for the control of migration of code in a concurrent distributed environment. We choose a subcalculus of the $lsd\pi$ calculus that fits the specifications of DiTyCO, a distributed implementations of TyCO, that lacks resource access control. We monitor three security policies: remote communication, process migration and channel creation. These policies correspond to the actions of the $lsd\pi$ calculus and enables us to control code migration. The security policies are defined at network level, 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. We start with a subset of $lsd\pi$ calculus, with a fixed number of sites, and present a non-trivial solution based on typing and subtyping relations to check that processes respect the security policies specified. It is our understanding that this work settle the ground basis for 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.

Acknowledgements

This work was partially supported by the Portuguese Fundação para a Ciência e a Tecnologia (via CLC and the project MIMO, POSI/CHS/39789/ 2001), and by the EU FEDER (via CLC) and the EU IST proactive initiative FET-Global Computing (projects Mikado, IST–2001–32222, and Profundis, IST–2001–33100).

References

1. G.Boudol, I. Castellani, F. Germain, and M. Lacoste. Models of distribution and mobility: State of the art. Mikado Deliverable D1.1.1, 2002.
2. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In J. Parrow G.J. Woeginger J.C.M. Baeten, J.K. Lenstra, editor, *Proc. of 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *LNCS*, pages 119–132. Springer-Verlag, 2003.
3. Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 2003. To Appear.
4. Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.
5. Luís Lopes, Álvaro Figueira, Fernando Silva, and Vasco T. Vasconcelos. A concurrent programming environment with support for distributed computations and code mobility. In *IEEE CLUSTER'00*, pages 297–306, 2000.

6. Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. Springer-Verlag, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, U. K., 1991.
7. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992. Available as Technical Reports ECS-LFCS-89-85 and ECS-LFCS-89-86, University of Edinburgh, U. K., 1989.
8. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 1(240):215–254, 2000.
9. Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
10. António Ravara, Ana Matos, Vasco T. Vasconcelos, and Luís Lopes. Lexically scoping distribution: what you see is what you get. In *FGC: Foundations of Global Computing*, volume 85(1) of *Electronic Notes in Theoretical Computer Science*, July 2003.
11. Vasco T. Vasconcelos, Luís Lopes, and Fernando Silva. Distribution and mobility with lexical scoping in process calculi. In *HLCL'98*, volume 16 (3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.