Typing migration control in $lsd\pi$.

Francisco Martins[*] António Ravara[†]

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

Framework. 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 [7, 8] 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.

DiTyCO, a distributed extension of the TyCO programming language [5, 12], 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 the 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, eliminating the costs of maintaining long remote sessions between clients and servers. The latest release of the language DiTyCO is available on the TyCO project site (http://www.ncc.up.pt/tyco/).

Aim. DiTyCO lacks a security mechanism to control the usage of important resources such as memory and *cpu* cycles. This work addresses such lacuna. Our main motivation is the control of code mobility by means of a simple, decidable, and low complexity type system. The system checks the integrity and the consistency of user-declared security policies, 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 between sites that know each other.

To focus on the problem, we develop our work using a lexically scoped distributed version of the pi-calculus—the $lsd\pi$ calculus [11]—that is the theoretical basis of DiTyCO. This calculus seems more suitable to study code migration in a distributed setting than other proposals, since the migration is triggered by channels themselves,

[*]Department of Mathematics, University of Azores, Portugal. E-Mail: fmartins@di.fc.ul.pt

[†]CLC and Department of Mathematics, Instituto Superior Técnico, Lisboa, Portugal. E-Mail: aravara@ist.utl.pt

rather than using explicit migration primitives (for an overview of distributed mobile calculi see a deliverable of an EU project [1]). 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. Indeed, DiTyCO is a good choice since it is a simple setting that lead us to a clear understanding of the security issues underlying code migration, and let us settle the basis for reasoning about resource usage.

To control the migration of processes between sites, we use three security policies, to control 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.

The defi nition 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 sites in the network that is being checked. Notice that we assume a closed world. This system is a tool to declare how code in other known sites may affect the computation of a given site, and to verify the compositionality of all these sites. 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.

Outline. Next section presents the syntax of the calculus, and of the type system. Section 3 is devoted to the operation semantics of the calculus. The main section of the paper is Section 4, where we introduce the type system, argue about some examples of networks that should be rejected by the type system, formalise a notion of runtime error suitable for our setting and purpose, and prove type safety. Section 5 concludes the paper, presents related work, and points directions for future research.

2 The calculus

This section briefly describes the calculus as well as its types, and some examples of networks, hinting informally the semantics of $lsd\pi$. A thorough presentation of the calculus is elsewhere [11].

Syntax. The $lsd\pi$ calculus extends the pi-calculus [8] distributing processes over flat networks of named sites. Communication occurs only within a site. Resources, located at creation time, maintain their location throughout the computation. 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 a channel may be addressed by its simple name—the channel—when it is at home, or by its located (or global) name—the pair channel–host site—whenever the reference is made from a foreign site.

The syntax of the calculus used in this paper 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*, $\mathcal{S}$, ranged over by r, s, t, disjoint from $\mathcal{C}$. *Channels*, may be simple—a—or located—$a@s$—, referring to a channel a from a site s. Let $\mathcal{N} = \mathcal{C} \cup \mathcal{S}$ be the set of the resource identifi ers. *Processes* are the standard π-calculus processes, apart from the input process, $u?(x : S)\ P$, as well as the replicated input, $u?{*}(x : S)\ P$, which mention a fi nite 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 fi rst class citizens, in the sense that 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.

simple channels $a, b, c, x, y, z \in \mathcal{C}$ sites $r, s, t \in \mathcal{S}$ sets of sites $S, R, T \subseteq \mathcal{S}$

channels $u ::= a \mid a@s$ values $v ::= () \mid u$

processes $P, Q ::= 0 \mid u!\langle v\rangle \mid u?(x:S)\ P \mid u?{*}(x:S)\ P \mid P \,|\, Q \mid (\boldsymbol{\nu}\, u)\ P$

networks $N, M ::= 0 \mid s_G[P] \mid N \,\|\, M \mid (\boldsymbol{\nu}\, a@s)\ N$

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

Figure 1: Syntax of $lsd\pi$.

Sites constitute the basic building blocks for networks. A network $s_G[P]$ denotes a site s, where a process P is running, with security policies bound by G. This set G defines the interactions allowed between s and the other sites in the surrounding network. It is intended to be written by the site administrator, who thus declares in this way the site security policies. The section on types presents these sets of security policies in detail. Networks are putted together using the *network parallel construct* $N \,\|\, M$. 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 remaining constructs allow us to restrict located channels in a network, $(\boldsymbol{\nu}\, a@s)\ N$, and 0 denotes an inactive network.

Notice the absence of an operator to create sites, which are thus constants. This scenario fits the present situation of DiTyCO. Allowing to create and pass around sites significantly complicates the technical work, since one has to take into account dynamic changes of site policies. An extension of the work reported here to incorporate this aspect has been developed and is presented elsewhere [6].

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

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

represents a network consisting of two sites r and s with security policies G_1 and G_2, respectively (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. In s the channel a declares which sites may remotely communicate with it and instantiate x. So, if r is in S, the process migrates first from r to s and then communicates with a within s. Notice that process $a@s!\langle b\rangle$ gives a clear understanding of the sites each name belongs to: the located channel $a@s$ is hosted by s; the simple channel b is from r. The above network reduces in two steps to

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

where r is now explicitly mentioned in the reference to channel b, since it "left home".

Types. Our approach consists in the specification of the security policies at site level, possibly 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.

A *typing* is a mapping from site names to pairs of site types and site policies. *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 $x\,!\,\langle v\rangle$ running at site s, we require that the sites of the channels that may instantiate x allow s to remote communicate with them. The val type has a unique value, $()$, and denotes a channel that carries no other channels.

The *tags*, i for *input*, o for *output*, and b for *both*, are part of a subtyping relation (inspired on the proposal of Pierce and Sangiorgi [10]) on the set of sites that may instantiate a given channel (defined in section 4). For example, this subtyping relation says that it is safe to use a channel of type $\mathrm{ch}(\gamma)@\{s, r\}^o$ whenever it is possible to use a channel of type $\mathrm{ch}(\gamma)@\{s\}^o$.

3 Semantics

The operational semantics of the calculus is presented following Milner *et al* [8]. We first define a *congruence relation* between processes and networks that simplifies the *reduction relation* introduced thereafter.

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

- At network level, the binding of a located channel, entails the binding of free occurrences of not only a located channel anywhere in the network, but also the occurrences as a simple channel at its host site.

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

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

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

A paper on $lsd\pi$ presents complete definitions and explanations on free names (function fn not shown in this paper), on bound names, and on substitution [11].

Structural congruence. The *structural congruence relation* is the least congruence relation closed under the rules in Figure 2 The first 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 $\mathsf{0}$ being the neutral element.

Scope extrusion rules, however, deserve a more detailed analysis. In the third rule of group 3, if we are creating a remote channel ($r \neq s$), then the remote site r must grant permission to create the name. (the set G_r defines the security policies for site r.) When we create a local name, then there must not exist a simple channel with the same name in P. A similar concern is expressed in forth rule of the same group. The reason for these side conditions becomes clear in the following examples. These two networks should not be in the congruence relation

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

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

Similarly,

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

1. $N \equiv M \qquad \text{if } N \equiv_\alpha M$
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. $((\boldsymbol{\nu}\, a@s)\ N) \parallel M \equiv (\boldsymbol{\nu}\, a@s)\ (N \parallel M) \qquad \text{if } a@s \notin \text{fn}(M)$

   $(\boldsymbol{\nu}\, a@r)\ (\boldsymbol{\nu}\, b@s)\ N \equiv (\boldsymbol{\nu}\, b@s)\ (\boldsymbol{\nu}\, a@r)\ N$

   $(\boldsymbol{\nu}\, a@r)\ s_G\boldsymbol{[}P\boldsymbol{]} \equiv s_G\boldsymbol{[}(\boldsymbol{\nu}\, a@r)\ P\boldsymbol{]} \qquad \text{if } (r \neq s \wedge s \in G_r(\text{new})) \vee (r = s \wedge a \notin \text{fn}(P))$

   $(\boldsymbol{\nu}\, a@s)\ s_G\boldsymbol{[}P\boldsymbol{]} \equiv s_G\boldsymbol{[}(\boldsymbol{\nu}\, a)\ P\boldsymbol{]} \qquad \text{if } a@s \notin \text{fn}(P)$
4. $s_G\boldsymbol{[}a@s\,!\,\langle v\rangle\boldsymbol{]} \equiv s_G\boldsymbol{[}a\,!\,\langle v\rangle\boldsymbol{]}, \quad s_G\boldsymbol{[}a@s?(x:S)\ P\boldsymbol{]} \equiv s_G\boldsymbol{[}a?(x:S)\ P\boldsymbol{]}$
5. $P \equiv Q \qquad \text{if } P \equiv_\alpha Q$
6. $((P \mid Q) \mid R) \equiv (P \mid (Q \mid R)), \quad (P \mid Q) \equiv (Q \mid P), \quad (P \mid \mathsf{0}) \equiv P$
7. $((\boldsymbol{\nu}\, u)\ P) \mid Q \equiv (\boldsymbol{\nu}\, u)\ (P \mid Q) \qquad \text{if } u \notin \text{fn}(Q)$

   $(\boldsymbol{\nu}\, u)\ (\boldsymbol{\nu}\, u')\ P \equiv (\boldsymbol{\nu}\, u')\ (\boldsymbol{\nu}\, u)\ P$
8. $(\boldsymbol{\nu}\, a)\ \mathsf{0} \equiv \mathsf{0}$

Figure 2: Structural congruence on processes and networks.

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 $lsd\pi$ calculus there is also a rule for splitting and regrouping sites: $s_G\boldsymbol{[}P\boldsymbol{]} \parallel s_G\boldsymbol{[}Q\boldsymbol{]} \equiv s_G\boldsymbol{[}P \mid Q\boldsymbol{]}$. However, in this version of the calculus, where sites are constant, the rule is covered by the migration rules RN-MIGI, RN-MIGO, and RN-MIGR ahead (see Figure 3).

Structural congruence on processes presents a similar set of rules when compared to structural congruence on networks. The only remark concerns rule 8 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 $\text{fn}((\boldsymbol{\nu}\, u)\ \mathsf{0}) \neq \text{fn}(\mathsf{0})$ when u is a located channel. Recall that s is free in $(\boldsymbol{\nu}\, a@s)\ \mathsf{0}$. Similar reasoning applies also to networks, where garbage collection would also have the malicious effect of erasing site policy annotations.

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

Definition 1 (Reduction contexts).

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

When moving a process from a site to another one, we need to translate the free channels of that process to take into account the location the process will end up in: those local to the source site become remote, and those belonging to the target site become simple as they reached home.

Definition 2 (Translation of identifiers). *Let $A \subseteq \mathcal{N}$, $A \cap \mathcal{C} = \{a_1, \ldots, a_n\}$ and $A \cap \mathcal{C}@r = \{b_1@r, \ldots, b_m@r\}$. Then,*

$$\sigma_{rs}(P, A) = P\{a_1@s/a_1, \ldots, a_n@s/a_n, b_1/b_1@r, \ldots, b_m/b_m@r\}\,.$$

Notation: Let $P\sigma_{rs}$ abbreviate the result of applying a name translation σ_{rs} to the process P, affecting its free names: $P\sigma_{rs} = \sigma_{rs}(P, \text{fn}(P))$.

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

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

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

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

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

$$\text{RP-CONT} \quad \frac{P \to Q}{E[P] \to E[Q]} \qquad \text{RP-STR} \quad \frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

$$\text{RN-SITE} \quad \frac{P \to Q}{s_G[\![P]\!] \to s_G[\![Q]\!]} \qquad \text{RN-CONT} \quad \frac{N \to M}{F[N] \to F[M]}$$

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

Figure 3: Reduction rules.

Definition 3 (Reduction relation). *The rules in figure 3 inductively define the reduction relation on $lsd\pi$ terms.*

Rule RP-COMM is the *communication* rule of the calculus. It is defined only locally and it is the standard asynchronous π-calculus communication rule. Rule RP-COMR defines the communication for replicated inputs. 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 channels lexically scoped, we use 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 that 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 download of code from a server site srv requested by client site cl. Assume that $r \notin \text{fn}(Q)$.

$$\begin{array}{l} cl[\![(\boldsymbol{\nu}\, req)\ dl@srv!\langle req\rangle \mid req!\langle\rangle \mid P]\!] \parallel \\ srv[\![(dl?{*}(r)\ r?()\ Q) \mid R]\!] \end{array}$$

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.

$$\begin{array}{rcr} cl[\![(\boldsymbol{\nu}\, req)\ dl@srv!\langle req\rangle \mid req!\langle\rangle \mid P]\!] \ \parallel\ srv[\![dl?{*}(r)\ r?()\ Q \mid R]\!] & \to & (\text{RN-STR, RN-MIGO}) \\ (\boldsymbol{\nu}\, req@cl)\ cl[\![req!\langle\rangle \mid P]\!] \ \parallel\ srv[\![dl?{*}(r)\ r?()\ Q \mid R \mid dl!\langle req@cl\rangle]\!] & \to & (\text{RP-COMR}) \\ (\boldsymbol{\nu}\, req@cl)\ cl[\![req!\langle\rangle \mid P]\!] \ \parallel\ srv[\![dl?{*}(r)\ r?()\ Q \mid req@cl?()\ Q \mid R]\!] & \to & (\text{RN-MIGI}) \\ (\boldsymbol{\nu}\, req@cl)\ cl[\![req!\langle\rangle \mid P \mid req?()\ Q\sigma_{srv,cl}]\!] \ \parallel\ srv[\![dl?{*}(r)\ r?()\ Q \mid R]\!] & \to & (\text{RP-COMM, RN-STR}) \\ cl[\![(\boldsymbol{\nu}\, req)\ Q\sigma_{srv,cl} \mid P]\!] \ \parallel\ srv[\![dl?{*}(r)\ r?()\ Q \mid R]\!] & & \end{array}$$

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 specified security policies.

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 two examples elucidate this situation.

Example 1. Consider the network $s_{\{\text{rem}:\{t\}\}}[P] \parallel r_{G_1}[a@s\,!\,\langle x\rangle]$. 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 site s fix the problem: $s_{\{\text{rem}:\{t,r\}\}}[P] \parallel r_{G_1}[a@s\,!\,\langle x\rangle]$.

Example 2. To type-check the network $s_{\{\text{rem}:\{r\}\}}[a?(x:\{r,t\})\ x\,!\,\langle c\rangle] \parallel r_{\emptyset}[a@s\,!\,\langle b\rangle]$ correctly, site s must be able to remote communicate both with site r, and site t. This is easily seen from the arguments of the input process at site s. Analysing the input continuation, $x\,!\,\langle c\rangle$, we conclude that the process could remote communicate with site r (and with site t as well), which does not concede any remote communication privilege at all. If r grants rem privileges to s, the network $s_{\{\text{rem}:\{r\}\}}[a?(x:\{r,t\})\ x\,!\,\langle c\rangle] \parallel r_{\{\text{rem}\{s\}\}}[a@s\,!\,\langle b\rangle]$ type-checks.

The control of code migration is performed using the policy keyword—**mig**—specifying which sites are allowed to upload code.

Example 3. The network $s_{\{\text{mig}:\{t\}\}}[P] \parallel r_{G_1}[a@s?(x:S)\ b\,!\,\langle x\rangle]$ is rejected because site s denies migration of code from site r as it is intended by process $a@s?(x:S)\ b\,!\,\langle x\rangle$. Including r in the policies of site s overcomes the problem: $s_{\{\text{mig}:\{t,r\}\}}[P] \parallel r_{G_1}[a@s?(x:S)\ b\,!\,\langle x\rangle]$

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

Example 4. The network $s_{\{\text{new}:\{t\}\}}[P] \parallel r_{G_1}[(\boldsymbol{\nu}\, a@s)\ a@s\,!\,\langle b\rangle]$ fails to type check because site s denies creation of remote channels (as well as remote communications) from site r. Regardless the actions of process P, that we are not taking into account for the example, the network $s_{\{\text{new}:\{t,r\},\text{rem}:\{r\}\}}[P] \parallel r_{G_1}[(\boldsymbol{\nu}\, a@s)\ a@s\,!\,\langle b\rangle]$ is well typed .

The following example shows a more trickier situation resulting from code migration.

Example 5. Consider the network $s_G[b@r?(x:S)\ a@s\,!\,\langle x\rangle] \parallel r_{\{\text{mig}:\{s\}\}}[0]$. The remote message $a@s\,!\,\langle x\rangle$ 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 $\text{rem}:\{s\}$ into the policies for site r.

A different kind of communication error arises 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 where these messages come from.

Example 6. The network $s_{\{\text{rem}:\{r\}\}}[a?(x:\{t\})\ 0] \parallel r_{G_1}[a@s\,!\,\langle b\rangle]$ 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 $x:\{t\}$). Network $s_{\{\text{rem}:\{r\}\}}[a?(x:\{t,r\})\ 0] \parallel r_{G_1}[a@s\,!\,\langle b\rangle]$ presents no security faults.

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

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

$$\gamma \leq \gamma \qquad \frac{\gamma_1 \leq \gamma_2 \quad \gamma_2 \leq \gamma_3}{\gamma_1 \leq \gamma_3} \qquad \frac{\gamma_1 \leq \gamma_2 \quad S^t \leq R^{t'}}{\mathrm{ch}(\gamma_1)@S^t \leq \mathrm{ch}(\gamma_2)@R^{t'}}$$

Figure 4: Subtyping relation.

SS-CHL $\Gamma \vdash_s a@r : \Gamma(r)_1(a)$ SS-CHS $\Gamma \vdash_s a : \Gamma(s)_1(a)$

Figure 5: Typing channels.

Subtyping. The binary relation $\leq$ on types is defined following Pierce and Sangiorgi [10], and is the least relation closed under the rules in figure 4. Intuitively, the subtyping relation allows for the inclusion of site identifiers (of where a channel can be located in) when we perform outputs, and allows for the exclusion of site identifiers when we perform inputs. If a channel is used both for input and output its type is fixed. For example, considering that $\gamma_1 \leq \gamma_2$, then $\mathrm{ch}(\gamma_1)@\{s, r, t\}^o \leq \mathrm{ch}(\gamma_2)@\{s, r\}^o$, and $\mathrm{ch}(\gamma_1)@\{s\}^i \leq \mathrm{ch}(\gamma_2)@\{s, r, t\}^i$.

Tags i, o, and b 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.

Typing. Figures 5, 6, and 7 present, respectively, the typing rules for channels, processes, and networks.

We record types for sites.[1] 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 5 for the typing rules for channels.

Judgements for processes, $\Gamma \vdash_{s,S} P$, besides the identity of the current site s, record 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, code migration, or channel creation) take place. We proceed by explaining the typing rules for processes that can be found in figure 6.

An 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 (2) 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

$$\frac{\begin{array}{cc}\Gamma \vdash_s v : \mathrm{ch}(\gamma)@\{s\}^b & \mathrm{ch}(\gamma)@\{s\}^b \leq \mathrm{ch}(\gamma)@\{s, r\}^i \\ \{t\} \subseteq \{s, t\} = \Gamma(r)_2(\mathrm{rem}) & \Gamma(r)_1(a) = \mathrm{ch}(\mathrm{ch}(\gamma)@\{s, r\}^i)@\{r\}^b\end{array}}{\Gamma \vdash_{s,\{t\}} a@r\,!\,\langle v\rangle}$$

On the other hand, if the output is performed over a simple channel 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 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). When typing a process $x\,!\,\langle v\rangle$ where the channel name x can be instantiated with either a simple channel or a located channel, one uses the rule SP-Outs. The sites that may send channels to instantiate x must be in the annotation of the input which introduces the parameter x.

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

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

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

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

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

$$\text{SP-NIL}\quad \Gamma \vdash_{r,S} 0 \qquad\qquad \text{SP-PAR}\quad \frac{\Gamma \vdash_{s,S} P \qquad \Gamma \vdash_{s,S} Q}{\Gamma \vdash_{s,S} (P \mid Q)}$$

$$\text{SP-RESS}\quad \frac{\Gamma(s)_1(a) = \mathrm{ch}(\gamma)@S^b \qquad \Gamma \vdash_{s,S} P}{\Gamma \setminus a@s \vdash_{s,S} (\boldsymbol{\nu}\, a)\ P} \qquad \text{SP-RESL}\quad \frac{\begin{array}{c}\Gamma(r)_1(a) = \mathrm{ch}(\gamma)@\{r\}^b \\ \Gamma \vdash_{s,S} P \qquad S \setminus s \subseteq \Gamma(r)_2(\mathrm{new})\end{array}}{\Gamma \setminus a@r \vdash_{s,S} (\boldsymbol{\nu}\, a@r)\ P}$$

Figure 6: Typing processes.

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. Notation $\Gamma \setminus x@s$ denotes the removal of channel x, located at site s, from the typing environment Γ. 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,

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

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

The typing rules for networks can be found in figure 7. 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 premises 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 more delicate than at site level, since we lack information about the site that has created the resource. 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.

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

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

$$\textsc{SN-Nil}\quad \Gamma \vdash 0 \qquad\qquad \textsc{SN-Par}\quad \frac{\Gamma \vdash N \qquad \Gamma \vdash M}{\Gamma \vdash (N \parallel M)}$$

$$\textsc{SN-Net}\quad \frac{\begin{array}{cc}\Gamma \vdash_{s,\{s\}} P & \Gamma(s)_2 = G\end{array} \\ \Gamma(s)_1(a) = \mathrm{ch}(\gamma)@\{s\}^b,\ \forall a \in \mathrm{dom}(\Gamma(s)_1)}{\Gamma \vdash s_G[P]}$$

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

Figure 7: Typing networks.

5 Results

On the type system. Subtyping is a preorder.

Lemma 1. *The relation $\leq$ is a preorder.*

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 2. *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 2, page 5), where the last typing rule applied is SP-RESL, to illustrate the use of the side condition $s \in G_1(\mathrm{new})$ when $a@s \notin \mathrm{fn}(P)$. 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 two subcases:

- $a@r \in \mathrm{fn}(P)$

  Therefore, by rule SP-RESL, $s \in \Gamma(r)_2(\mathrm{new})$ and so, by SP-RESL and SN-NET, we prove that $\Gamma \vdash s_G[(\boldsymbol{\nu}\, a@r)\ P]$.

- $a@r \notin \mathrm{fn}(P)$

  In this case $s \notin S$, but the side condition $s \in G_1(\mathrm{new})$ 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 $s = r$ is not relevant because it is always possible 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 2 (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. (sketch) 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. ☐

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

Definition 4 (runtime errors). *Assume that $r \neq s$. Let $\mathcal{E} = \{N | N \rightarrow^\star \nu u_1 \ldots \nu u_k(M' \parallel M)\}$, and M of the form*

$$
\begin{aligned}
r_{G_1}\textbf{[}P\textbf{]} \parallel s_{G_2}\textbf{[}a@r\,!\,\langle v \rangle\textbf{]}, &\qquad s \notin G_1(\mathrm{rem}) &(1)\\
r_{G_1}\textbf{[}P\textbf{]} \parallel s_{G_2}\textbf{[}a@r?(x:T)\ P\textbf{]}, &\qquad s \notin G_1(\mathrm{mig}) &(2)\\
s_G\textbf{[}(a?(x:T)\ P) \mid a\,!\,\langle b@r \rangle\textbf{]}, &\qquad r \notin T &(3)\\
r_{G_1}\textbf{[}P\textbf{]} \parallel s_{G_2}\textbf{[}(\boldsymbol{\nu}\, a@r)\ P\textbf{]}, &\qquad s \notin G_1(\mathrm{new}) &(4)
\end{aligned}
$$

Networks that violate a security policy are in the set $\mathcal{E}$. The following result states that well-typed networks do not belong to $\mathcal{E}$.

Theorem 3 (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. □

6 Conclusions

Summary. We present a type system to control the migration of code in a concurrent distributed calculus where a network has a fi xed number of sites that know each other. This setting fi ts DiTyCO, a distributed implementations of the language TyCO, which presently lacks resource access control. 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 defi ned by the site administrators, following an intuitive and easy approach. For each site, its administrator specifi es what operations other sites are allowed to perform. The type system checks if the processes running at those sites respect its security policies, and then checks if all the sites in the network will interoperate without violating each other policies. Specifi cally, we prove subject reduction, defi ne runtime errors, and then prove type safety.

The current setting allow us to focus on the security policies for resources. We start with a subset of $lsd\pi$ calculus, with a fi xed number of sites, and 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. We prove subject reduction and type safety.

Further work. It is our understanding that this work settles the ground basis for further developments along two main directions: (a) the defi nition of security policies at resource level and therefore be able to refi ne the interaction between sites; (b) and the ability to adjust security policies dynamically.

We have already extended this system to deal with an arbitrary number of sites in networks with dynamic topology [6].

Related work. Other approaches to resource security in distributed mobile calculi comprise DPI [3, 4] and Klaim [2, 9]. See [1] for a general survey on concurrent mobile calculus, type systems, and security policies. DPI possesses an explicit objective construct to 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 diffi cult 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 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 *Proc. of 30th ICALP'03*, volume 2719 of *LNCS*, pages 119–132. Springer-Verlag, 2003.

[3] M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 2003.

[4] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.

[5] L. Lopes, A. Figueira, F. Silva, and V. Vasconcelos. A concurrent programming environment with support for distributed computations and code mobility. In *IEEE CLUSTER'00*, pages 297–306, 2000.

[6] F. Martins and V. Vasconcelos. Controlling security policies in a distributed environment. DI/FCUL TR 04–1, Department of Informatics, University of Lisbon, April 2004.

[7] R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. Springer-Verlag, 1993.

[8] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992.

[9] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 1(240):215–254, 2000.

[10] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[11] 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.

[12] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and mobility with lexical scoping in process calculi. In *HLCL'98*, volume 16 (3) of *ENTCS*. Elsevier Science Publishers, 1998.