MIKADO Global Computing Project IST-2001-32222

Mobile Calculi Based on Domains

Type Systems for Open Networks, using the MIKADO core programming models

Final Deliverable D2.1.4

Title	:	Type Systems for Open Networks, using the MIKADO core programming models
Author(s)	:	D. Gorla (Univ. of Florence)
Abstract	:	R. De Nicola, D. Gorla and R. Pugliese (Univ. of Florence) M. Coppo, M. Dezani and E. Giovannetti (Univ. of Torino) F. Martins and V. Vasconcelos (Univ. of Lisbon)
Status	:	Deliverable D2.1.4, Public
Report No	:	RR/WP2/4
Date	:	February 2005

Abstract

In the last decade, several foundational formalisms for global computing have appeared in literature to improve the understanding of the complex mechanisms underlying such new computational scenario. In their design, the integration of security mechanisms is a major challenge and great efforts have been recently devoted to embed such mechanisms within standard programming features.

In this deliverable, we shall focus on the security mechanisms put forward by *type systems*, that are used for expressing and checking behavioural properties concerning mobility, resource access, security, etc. Moreover, when dealing with distributed and mobile computing in wide-area "open" systems, one is often confronted with a scenario where interaction may take place between parties whose respective properties are unknown or only partially known to each other. If stopping the execution for re-checking is to be avoided, each component must dynamically carry with it sufficient behavioural information that can be checked at runtime by the other ones interacting with it.

We present here three typing approaches focussed on security properties of open systems, and on their enforcement mechanisms. The solutions proposed do not rely on a single process calculus but encompass three successfull models for global computing systems: Dp, Ambient and KLAIM.

Contents

1 Introduction

The spreading of small, powerful portable machines like PDAs, cellular phones, and laptop computers, equipped with long lasting batteries and wireless communications, is promoting the integration of a broad range of services and encouraging the sharing of resources. Consequently, the protection of personal data and resources from being abusively used is a central concern for the global network participants. This has prompted the design of new theoretical models of computing: in particular, distributed process calculi and ambient calculi, for example [6, 14, 34, 37].

These calculi have greatly improved the formal understanding of the complex mechanisms underlying global computing but, although share similar intentions and motivations, rest on different design choices. Usually, they permit describing both single *p* o , and a of located, possibly migrating, processes. Nets are collections of d that can be thought of as physically distributed machines or as logical partitions of the same machine. Each node is referrable via an address, is connected to other nodes and hosts a set of processes. Depending on the design choices of the calculus, e.g. nets may be plain or hierarchically structured, and the notions of process, node and net may collapse. In the most basic setting, processes are built up from the empty process and from the basic b by using standard operators, e.g., action prefixing, name restriction, parallel composition and replication. The basic actions permit data exchange, spawning of new, possibly remote, processes and creation of new resources.

In the design of programming languages for global computing, the integration of security mechanisms is a major challenge and great efforts have been recently devoted to embed such mechanisms within standard programming features. Several language-based security techniques have been proposed that range from type systems [6, 13, 28, 34], to data flow analysis [4, 22, 30, 45], from in-lined reference monitoring [24] to proof-carrying code [44]. We refer the reader to [48] for an overview of some of these techniques.

The major goal of language-based security is to design languages that are flexible, expressive and safe. Unfortunately, these are often contrasting requirements. For example, mobile code deeply increases flexibility and thus expressivity of programming languages, but introduces new security problems related to unwanted accesses to classified data. Indeed, when programming has to take into account networks with mobile agents, existence in the environment of malicious principals, that can put security of data at risk, must be assumed. Malicious nodes can attack a mobile process and compromise its integrity through code modification or its secrecy through leakage of sensitive data. But one has also to take into account existence of malicious mobile processes that might attempt to access or forge private data of the network nodes hosting them.

A programming language for global computing should thus be equipped with a foundational model that also encompasses security features; the proof that an application is 'safe' could then be done by relying on formal methods. In our view, the language security model should consider existence of misbehaving entities in the execution environment of applications. Moreover, only local knowledge of the environment can be assumed because it would be impossible to collect global information in a network with possibly malicious nodes under the control of thousands different administrative authorities.

In this deliverable, we shall focus our attenction on the security mechanisms put forward by type systems, that are used for expressing and checking behavioural properties concerning mobility, resource access, security, etc. In most of them, a system or component is represented by a term t of a given calculus, a type T assigned to t, and an environment Γ. In the standard view, as is well-known, the term t abstractly describes the implementation, its type T may express some behavioural properties, and the environment Γ is a set of assumptions on the outside world. Typically, these are assumptions on types of non-local names; there is thus the notion of a global environment, which is the abstract description of situations where all the interacting parties are known in advance to each other, so that static checks performed before execution ensure the correctness of the whole system.

When dealing with distributed and mobile computing in wide-area "open" systems, however, one is often confronted with a scenario where interaction may take place between parties whose respective properties are

unknown or only partially known to each other. If stopping the execution for re-checking is to be avoided, each component must dynamically carry with it sufficient behavioural information that can be checked at runtime by the other ones interacting with it. This may correspond to a formal system where, like in the one proposed in [5], the typing judgment $\Gamma \vdash_a t{:}T$ is relative to a locality a, and may be "packed" into a new kind of term $a[t]_T^\Gamma$ that carries at runtime the typing information.

6 e f h d v h This deliverable is logically organised in three parts, each one corresponding to an independent work. Although the underlying calculi are different and the aims of the type systems presented in each setting differ, the general purpose of all these works is the same and, thus, they can be naturally merged together in this deliverable.

We start in Section 2 with the presentation of the work in [41]. Here we have a type system to control the migration of code between network nodes in a concurrent distributed framework, using the language Dπ [34]. Resource access policies are expressed as types and enforced via a type system. Types describe paths travelled by migrating code, enabling the control of history sensitive access to resources. Sites are logically organised in subnetworks that share the same security policies, statically specified by a network administrator. The type system guarantees that well-typed networks are exempt from security policies violations at runtime.

Then, in Section 3 we summarise the work in [16]. Here, an ambient calculus with both static and dynamic types is presented, where the latter ones represent mobility and access rights that may be dynamically consumed and acquired in a controlled way. Novel constructs and operations are provided to this end. Type-checking is purely local, except for a global hierarchy that establishes which locations have the authority to grant rights to which: there is no global environment (for closed terms) assigning types to names. Each ambient or process move is subject to a double authorization, one static and the other dynamic: static type-checking controls (communication and) "active" mobility rights, i.e., where a given ambient or process has the right to go; dynamic type-checking controls "passive" rights, i.e., which ambients a given ambient may be crossed by and which processes it may receive.

Finally, in Section 4 we present the work in [21]. Here, a programming notation is introduced that can be used for protecting secrecy and integrity of data in global computing applications. The approach is based on the explicit annotations of data and network nodes. Data are tagged with information about the allowed movements, network nodes are tagged with information about the nodes that can send data and spawn processes to them. The annotations are used to *o a* movements of data and processes. The approach is illustrated by applying it to three paradigmatic calculi for global computing, namely CKLAIM (a calculus at the basis of KLAIM), Dπ (a distributed version of the π-calculus) and Mobile Ambients Calculus. For all of these formalisms, it is shown that their semantics guarantees that computations proceed only while respecting confinement constraints. Namely, it is proven that, after successful static type checking, data can reside at and cross only authorised nodes. "Local" formulations of this property where only relevant sub-nets type check are also presented. Finally, the theory is tested by using it to model secure behaviours of a UNIX-like multiuser system.

Comparison with related work is relegated to the end of each section; this simplifies the comparison of each approach with more strictly related work.

2 His t ory-bas eAd ce s Cont rd for Dis t rib e dPro ce s s e s

This section proposes a discipline to control the security of resources in a mobile distributed environment. Take, for example, a typical network architecture for an institution that exposes some of their servers (*eg* . SSH, HTTP, SMTP, and DNS) to an untrusted network, like the Internet, as described in figure 1 (cf. [52]). The task of the network administrator is to find the correct balance between hiding and revealing the institution's services to the outside world. Some institutions, however, need to give permission to untrusted third

d)

Policy inheritance The inheritance of security policies helps in designing and maintaining policies for subgroups. The inheritance is twofold: (a) explicit, via keyword inherit, stating that a subgroup inherits the security policies of its direct parent groups; (b) implicit, adopting the identity of a parent group. Defining

$v ::=$	*Value*	$n ::=$	*Name*
$a_{@s}$	located channel	a, b, c, x	channels
$\mid \diamond$	basic value	$\mid r, s, t, y$	sites
		$\mid f, g, h$	groups
$P, Q ::=$	*Process*	$N, M ::=$	*Network*
stop	termination	stop	termination
$\mid (\nu n\colon L_{@s})\, P$	restriction	$\mid (\nu n\colon L)\, N$	restriction
$\mid P \mid Q$	composition	$\mid N \mid M$	composition
$\mid$ goto $s.P$	migration	$\mid s[P]$	site
$\mid a!\langle v \rangle$	output		
$\mid a?(v)\, P$	input	see figure 11 for	the syntax of types L
$\mid a?*(v)\, P$	replication		

Figure 2: Syntax of Dπ.

1. $((N \mid M) \mid M') \equiv (N \mid (M \mid M'))$ $\quad (M \mid N) \equiv (N \mid M)$ $\quad (N \mid \mathsf{stop}) \equiv N$
2. $(\nu n\colon T)\, N \mid M \equiv (\nu n\colon T)\, (N \mid M)$ if $n \notin \mathrm{fn}(M)$
   $(\nu n\colon T)\, (\nu m\colon T')\, N \equiv (\nu m\colon T')\, (\nu n\colon T)\, N$ if m not in T, and n not in T'
   $(\nu n\colon L_{@s})\, s[P] \equiv s[(\nu n\colon L)\, P]$ if $n \neq s$
3. $s[P] \mid s[Q] \equiv s[P \mid Q]$
4. $(\nu n\colon T)\, \mathsf{stop} \equiv \mathsf{stop}$ $\quad (\nu s\colon T)\, s[\mathsf{stop}] \equiv \mathsf{stop}$

Figure 3: Structural congruence.

2.1 Dπ syntax and operational semantics

This section deals with the syntax and the operational semantics of Dπ, mainly taken from Hennessy and Riely [34].

Syntax The syntax of the calculus is defined in figure 2. The main difference w.r.t. the original Dπ is the usage of groups, namely the new constructor to create groups. We consider a monadic version of the calculus where only located names can be passed around, since our main focus is the control of migration, not that of communication.

We briefly address the Dπ syntax; the interested reader should refer to [32, 34] for motivations and details. The calculus presents two main syntactic categories: processes and networks. At process level we find the usual asynchronous π-calculus constructs [2, 35]; processes are built from the inactive process, stop, and from the asynchronous output process, $a!\langle v \rangle$, using three constructs: name restriction, $(\nu n\colon T)\, P$, parallel composition, $P \mid Q$, and input, $a?(v)\, P$. We also include a form of replicated input, $a?*(v)\, P$. Moreover, Dπ contains an operator that sends a process P to a specific location s: the goto $s.P$ process.

Networks are assembled from the inaction network, stop, and from processes running at specific named locations called sites, $s[P]$, using name restriction, $(\nu n\colon T)\, N$, and parallel composition, $N \mid M$.

Bound names The binders of the calculus are the usual in π-calculi like languages: name n is bound in $(\nu n\colon T)\, P$ and in $(\nu n\colon T)\, N$, whereas x and y are both bound in $a?(x_{@y})\, P$. Networks are taken up to α-congruence in such a way that bound names are different from free names and from each other.

Operational semantics is defined on top of a *structural congruence* relation, $\equiv$, that is the least congruence relation closed under the rules defined in figure 3. It follows closely the structural congruence relation introduced for Dπ.

Reduction in figure 4 is mainly taken from Dπ, except for obvious adjustments to incorporate groups.

$$s[a!\langle b@r\rangle] \mid s[a?(x@y)\,P] \to s[P\{r/y\}\{b/x\}] \qquad \text{COMC}_1$$
$$s[a!\langle\diamond\rangle] \mid s[a?(\diamond)\,P] \to s[P] \qquad \text{COMC}_2$$
$$s[a!\langle b@r\rangle] \mid s[a?*(x@y)\,P] \to s[P\{r/y\}\{b/x\}] \mid s[a?*(x@y)\,P] \qquad \text{COMR}_1$$
$$s[a!\langle\diamond\rangle] \mid s[a?*(\diamond)\,P] \to s[P] \mid s[a?*(\diamond)\,P] \qquad \text{COMR}_2$$

$$s[\text{goto } r.P] \to r[P] \qquad \frac{N \to M}{(\nu n{:}\, L@s)\,N \to (\nu n{:}\, L@s)\,M} \qquad \text{MIG, RES}$$

$$\frac{N \to N'}{N \mid M \to N' \mid M} \qquad \frac{N \equiv N' \quad N' \to M' \quad M' \equiv M}{N \to M} \qquad \text{PAR, STR}$$

Figure 4: Reduction rules.

$\mathcal{P}$ ::=	*Policy*	τ ::=	*Effect*	S ::=	*Path*
$\{\pi_1 \ldots \pi_n\}$		useRes	output	ε	empty path
		| installRes	input	| g	group
π ::=	*Rule*	| createRes	ch. creation	| $\cdot$	any group
$\tau{:}\, S$	effect rules	| createSite	site creation	| SS	concatenation
| forward: S	code forward	| createGroup	group creation	| $S + S$	alternation
| inherit	inherit policies			| $S\star$	kleene star

Figure 5: Syntax for security policies.

2.2 Security policy

Policy, rule, path A security policy ($\mathcal{P}$) consists of a set of rules (π). Effect rules ($\tau{:}\, S$) describe the set of admissible paths in the group hierarchy that code must visit before being able to perform the action the policy protects. Rule forward governs the migration of code. For code migration to succeed, there must be a path all along the group hierarchy that authorises the forwarding of the code to the destination site. The inherit allows a group to import the rules defined for its direct parents.

The *effects*, τ, correspond directly to the actions of the calculus: the input and output actions are related with the installRes and useRes effects, respectively; channel, site, and group creation are associated with createRes, createSite, and createGroup effects, respectively.

A *path*, S, is a regular expression. A group g stands for itself, the symbol $\cdot$ is a wild card that represents any group. Concatenation, alternation, and Kleene closure possess the usual meaning.

Environment, type, and well-formedness A *type environment* Γ is a partial function of finite domain from names to types. For the current section we outline that the type for sites ($s{:}\, G$) is the set of groups that the site belongs to, and the type for groups ($g{:}\, (\mathcal{P}, G)$) is a pair: security-rules, parent-groups. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ. When $x \notin \mathrm{dom}(\Gamma)$, we write $\Gamma, x{:}\, T$ for the type environment Γ' such that $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = T$ and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$. We use $\tilde{X}$ to denote a possibly empty sequence of X's. Similarly, $r{:}\, F$ and $g \in G$ denote $r_1{:}\, F_1, \ldots, r_n{:}\, F_n$ and $g_1 \in G_1, \ldots g_n \in G_n$, respectively.

Functions allows and canEnter, defined in figures 6 and 7, perform security checking. Before outlining function allows, we give an overview of function matches defined in [41]. A formula e matches S means that a path e fits in path pattern S. The rules for most path constructs are straightforward; we present the

$$\frac{\Gamma(g) = (\mathcal{P} \cup \{\tau : S\}, G) \quad \Gamma \vdash \mathfrak{p} \text{ matches } S \quad \tau \neq \text{createGroup}}{\Gamma \vdash g \text{ allows } \mathfrak{p} : \tau}$$

$$\frac{\Gamma(g) = (\mathcal{P} \cup \{\text{createGroup} : S\}, G) \quad \Gamma \vdash \mathfrak{p} \text{ matches } S \quad \forall h \in G \quad \Gamma \vdash h \text{ allows } \mathfrak{p} : \{\text{createGroup}\}}{\Gamma \vdash g \text{ allows } \mathfrak{p} : \{\text{createGroup}\}}$$

$$\frac{\Gamma(g) = (\mathcal{P} \cup \{\text{forward} : S\}, \emptyset) \quad \Gamma \vdash \mathfrak{p} \text{ matches } S}{\Gamma \vdash g \text{ allows } \mathfrak{p} : \{\text{forward}\}}$$

$$\frac{\Gamma(g) = (\mathcal{P} \cup \{\text{forward} : S\}, \{h\} \cup G) \quad \Gamma \vdash \mathfrak{p} \text{ matches } S \quad \Gamma \vdash h \text{ allows } \mathfrak{p} : \{\text{forward}\}}{\Gamma \vdash g \text{ allows } \mathfrak{p} : \{\text{forward}\}}$$

$$\frac{\Gamma(g) = (\{\text{inherit}\} \cup \mathcal{P}, \{h\} \cup G) \quad \Gamma \vdash h \text{ allows } \mathfrak{p} : A}{\Gamma \vdash g \text{ allows } \mathfrak{p} : A}$$

$$\frac{\forall g \in G, \forall \widetilde{f \in F}, \forall \pi \in A,\ \Gamma \vdash g \text{ allows } \mathfrak{p} : \pi}{\Gamma \vdash G \text{ allows } \mathfrak{p} : A}$$

$$\frac{\Gamma, s : G, r : F \vdash G \text{ allows } \mathfrak{p} : A}{\Gamma, s : G, r : F \vdash s \text{ allows } \mathfrak{p} : A}$$

$$\frac{}{\Gamma \vdash s \text{ allows } s : A}$$

Figure 6: allows relation.

$$\frac{\Gamma(f) = (\mathcal{P}, \emptyset)}{\Gamma \vdash \mathfrak{g} \text{ canEnter } f}$$

$$\frac{\Gamma(f) = (\mathcal{P}, \{h\} \cup G) \quad \Gamma \vdash h \text{ allows } \mathfrak{g} : \text{forward}}{\Gamma \vdash \mathfrak{g} \text{ canEnter } f}$$

$$\frac{}{\Gamma \vdash s \text{ canEnter } s}$$

$$\frac{\forall \widetilde{g \in G}, \forall f \in F, \Gamma \vdash \mathfrak{g} \text{ canEnter } f}{\Gamma \vdash \mathfrak{G} \text{ canEnter } F}$$

$$\frac{\Gamma, s : G, r : F \vdash \mathfrak{G} \text{ canEnter } H}{\Gamma, s : G, r : F \vdash \mathfrak{s} \text{ canEnter } r}$$

Figure 7: canEnter relation.

non-standard ones: a group matches itself or any group in its hierarchy.

$$\Gamma \vdash g \text{ matches } g \qquad \frac{\Gamma, g : (\mathcal{P}, G) \vdash h \text{ matches } f \quad h \in G}{\Gamma, g : (\mathcal{P}, G) \vdash g \text{ matches } f}$$

A formula g allows $\mathfrak{p} : \pi$ (figure 6) says that group g allows code that travelled through path $\mathfrak{p}$ to perform action π; path $\mathfrak{p}$ is matched against the path pattern associated with policy π using function matches. We use A in the rules to denote a set of effects τ. The createGroup effect receives special treatment, since the creation of a group establishes a new node in the group hierarchy, and the groups above must accept its new member. Since a group may identify itself as any of its parents, the creation of a subgroup must collect the acceptance of the whole hierarchy. Forwarding code requires that at least one branch in the hierarchy grants the forward policy to the path the code travelled. When the inherit policy keyword is set for a group, the security policies for the direct parent groups are considered as a part of the security specification for the group.

Function canEnter (figure 7) checks whether code that travels through a given path has permission to enter a target group. A formula $\mathfrak{g}$ canEnter f means that group f accepts code that has travelled through

[illegible] site [illegible] restriction [illegible]

$$s[P]^{\tilde{r}}_{\Gamma} \qquad (\nu_{\tilde{t}}\, n\colon T\,)\, N$$

[illegible]

2. $(\nu_{\tilde{t}}\, n\colon L@s\,)\, s[P]^{\tilde{t}}_{\Gamma \sqcap n\colon L@s} \equiv_T s[(\nu n\colon L\,) P]^{\tilde{t}}_{\Gamma}$ if $n \notin \mathrm{dom}(\Gamma) \cup \{s\}$
3. $s[P]^{\tilde{t}}_{\Gamma} \mid s[Q]^{\tilde{t}}_{\Gamma} \equiv_T s[P \mid Q]^{\tilde{t}}_{\Gamma}$

Figure 8: The tagged language – syntax and structural congruence.

$$s[a!\langle b@r\rangle]^{\tilde{t}}_{\Gamma;r\colon G} \mid s[a?(x@y)\,P]^{\tilde{u}}_{\Delta} \mapsto s[P\{r/y\}\{b/x\}]^{\tilde{u}}_{\Delta \sqcap r\colon G} \qquad \text{(T-COMC}_1\text{)}$$

$$s[a!\langle \diamond\rangle]^{\tilde{t}}_{\Gamma} \mid s[a?(\diamond)\,P]^{\tilde{u}}_{\Delta} \mapsto s[P]^{\tilde{u}}_{\Delta} \qquad \text{(T-COMC}_2\text{)}$$

$$s[a!\langle b@r\rangle]^{\tilde{t}}_{\Gamma;r\colon G} \mid s[a?{*}(x@y)\,P]^{\tilde{u}}_{\Delta} \mapsto s[P\{r/y\}\{b/x\}]^{\tilde{u}}_{\Delta \sqcap r\colon G} \mid s[a?{*}(x@y)\,P]^{\tilde{u}}_{\Delta} \qquad \text{(T-COMR}_1\text{)}$$

$$s[a!\langle \diamond\rangle]^{\tilde{t}}_{\Gamma} \mid s[a?{*}(\diamond)\,P]^{\tilde{u}}_{\Delta} \mapsto s[P]^{\tilde{u}}_{\Delta} \mid s[a?{*}(\diamond)\,P]^{\tilde{u}}_{\Delta} \qquad \text{(T-COMR}_2\text{)}$$

$$s[\text{goto } r.P]^{\tilde{t}}_{\Gamma} \mapsto r[P]^{\widetilde{st}}_{\Gamma} \qquad \frac{N \mapsto M}{(\nu_{\tilde{t}}\, n\colon T\,)\, N \mapsto (\nu_{\tilde{t}}\, n\colon T\,)\, M} \qquad \text{(T-MIG, T-RES)}$$

([illegible] PAR, [illegible] STR [illegible] 4)

Figure 9: The tagged language—reduction.

path [illegible]. This privilege is controlled using the forward policy. Code that went all along path [illegible] is able to enter the frontier of group f, if there exists a path through f's hierarchy granting, at each group in the path (except for the target group), the forward right to [illegible].

T[illegible]. To precise the notion of runtime errors we need to make explicit in the syntax the path the code travelled and the group policies (cf. [34]). Therefore, we present a tagged version for the language introduced in figure 2 and the corresponding tagged semantics.

T[illegible] Figure 8 (syntax) summarises the syntactic changes: (a) we append to the site constructor the [illegible] the code has visited and the set of assumptions needed to check security policies for the processes it runs; (b) we add to name restriction the sequence of sites followed by the code before creating the name, since, by scope extrusion, a name may appear at network level and we need this information to formalise runtime errors. All the remaining syntax is left unchanged.

T[illegible] Name extrusion (rule 2, figure 8) records the code journey leading to name creation, whereas restricting it to a site is only possible when the code running at the site followed the same migration path as that of the name creation. Notice that the set of assumptions at the left-hand side of the congruence relation enlarges with the name declared at network level, announcing the creation of the name. The meet operator, $\sqcap$, (cf. [32, 34]), used to combine the new name with that in type assumptions is defined in [41]. Merging sites (rule 3, figure 8) is only viable when the tagged information agree: it is not possible to merge sites that execute code travelling through different locations or that are governed by distinct security policies. The remaining structural congruence rules reflect the syntactic adjustments and are omitted.

T[illegible] (fig[illegible] The main differences w.r.t.untagged reduction (figure 4) concern communication and code migration. Communication may occur under dissimilar views of the security policies of a site, in particular, the input process may not use or even have knowledge of the value it is going to receive (except for its type). Therefore, communication updates the typing assumptions of the receiving process with information from the emitting process. Rule T-COMC$_1$ updates the resulting process with type informa-

R-OUT	$s[a!\langle v\rangle]^{\widetilde{r}}_{\Gamma} \overset{err}{\longmapsto}$	if $\Gamma \not\models s$ allows $\mathbf{e}$: useRes
R-INP	$s[a?(v)P]^{\widetilde{r}}_{\Gamma} \overset{err}{\longmapsto}$	if $\Gamma \not\models s$ allows $\mathbf{e}$: installRes
R-MIG	$s[\mathsf{goto}\, t.P]^{\widetilde{r}}_{\Gamma} \overset{err}{\longmapsto}$	if $\Gamma \not\models s\mathbf{e}$ canEnter t
R-RES$_1$	$s[(\boldsymbol{\nu} a\colon L)P]^{\widetilde{r}}_{\Gamma} \overset{err}{\longmapsto}$	if $\Gamma \not\models s$ allows $\mathbf{e}$: createRes
R-RES$_2$	$(\boldsymbol{\nu}_{\widetilde{r}} a\colon L@s)N \overset{err}{\longmapsto}$	if $\Gamma \not\models s$ allows $\mathbf{e}$: createRes

Figure 10: Runtime errors.

tion for the communicated site r. When site r is not mentioned in process P, the type information is just appended to typing Δ, otherwise we compute the least common supertype of the type figuring in Δ and the one communicated. The subject reduction theorem (page 15) guarantees that for well-typed processes the meet operation is always defined. The remaining rules are either similar to the above (T-COMR$_1$) or close to their untagged version (T-COMC$_2$ and T-COMR$_2$).

As for code migration, rule T-MIG, we append the name of the source site to the migration path. This information is fundamental to reason about security. We check the security policies considering the sites that migration code visits because this information is important to express the trust between the destination group and the rest of the network. Rule T-RES results from the syntax update.

Runtime errors. The unary relation, $\overset{err}{\longmapsto}$, defined in figure 10, identifies processes that break some security policy during reduction.

The output (input) process fails, R-OUT (R-INP), if the site that sent the code, r, has no permission to use (install) resources. We omit the rule for replicated input, since it is similar to rule R-INP. For code migration, rule R-MIG states that a goto process incurs in a runtime error if it cannot enter the border of the groups where the target site resides. Notice the role of typing Γ—a placeholder for security policies—, and the need to talk about the site where the code is, s, the sequence of sites visited by the code, $\mathbf{e}$, and the site where the code is migrating to, t. Rule R-RES$_1$ says that the channel creation operation fails if the current site does not allow the site that sent the code to create channels. Rule R-RES$_2$ is similar.

2.3 Typing system

In this section we present two type and effect systems (for the tagged and for the untagged languages) that check whether networks respect the security policies defined for groups. The type systems are based on a subtyping relation à la Sangiorgi and Pierce [46], and are parametric in two functions that are used to check the security policies, namely, the allows and the canEnter functions.

Types. The syntax for types is depicted in figure 11. We assign types to channels, to sites, and to groups. Types, T, may be local or global: local types, L, are used when creating names at a given site; global types (or located types), $L@s$, are assigned to names when declared at network level.

Channel types, L, aggregate channel types, C, that trace the type of the values that are communicated along the channel, as well as its usage (input, output, or both); site types, G, that simply record the set of groups the site belongs to; and group types, $(\mathcal{P}, G)$, which is a central notion in our work: it is at group level that we record information for security, namely, (a) the set of policies, $\mathcal{P}$, that govern the interaction with the network, and (b) the set, G, of groups that the group is allowed to enter.

Channels can carry other channels, as well as basic values, as described by value types, V. The type of channels that can be carried assumes the form $C@G$, where C is the type of the channels that can be carried, and G is the set of groups hosting the communicated channels. The subtype relation characterising channel types using I is introduced in figure 12.

Subtyping. The subtyping relation, $<:$, is defined as the least preorder relation on types that satisfies the rules in figure 12 where channels are tagged according to their usage: input (r), output (w), and input/output

$T ::=$	*Type*	$C ::=$	*local channel*	$V ::=$	*value type*
L	local type	$\langle V \rangle^I$	local channel	$C_{@}G$	channel
$\mid L_{@}s$	global type			$\mid$ unit	basic type
$L ::=$	*local type*	$I ::=$	*tag*	G	$\varepsilon \mid \rho \mid g \mid \rho$
C	local channel	r	input		
$\mid G$	site type	$\mid$ w	output		
$\mid (\mathcal{P}, G)$	group type	$\mid$ rw	input/output		

Figure 11: Syntax of types.

$$\mathsf{unit} <: \mathsf{unit} \qquad C_1 <: C_2 \quad G_1 \subseteq G_2 \; C_{1@}G_1 <: C_{2@}G_2 \qquad \textit{value subtyping}$$

$$\frac{i = \mathsf{r}, \mathsf{rw} \quad V_1 <: V_2}{\langle V_1 \rangle^i <: \langle V_2 \rangle} \qquad \frac{i = \mathsf{w}, \mathsf{rw} \quad V_2 <: V_1}{\langle V_1 \rangle^i <: \langle V_2 \rangle} \qquad \frac{V_1 <: V_2 \quad V_2 <: V_1}{\langle V_1 \rangle \quad <: \langle V_2 \rangle}$$

local channel subtyping

$$\frac{C_1 <: C_2}{C_{1@}s <: C_{2@}s}$$

located channel subtyping

Figure 12: Subtyping relation.

(rw). We extend the subtyping relation to deal with types involving groups. 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. The subtyping rules are straightforward. Notice the set inclusion to handle groups in value subtyping and the last subtyping rule that relates located channels.

Typing the effects of actions The type system collects the effects of the actions performed by processes. For instance, process $a!\langle b_{@}r \rangle$ running at site s has effect useRes. Then, following to a goto action we check whether the path travelled by the code has the right privileges to perform the intended action, in the present case an *input*. At network level there is nothing to be checked, since there is no computation taking place. Also, we do not check code running at its host site—code that is not in the continuation part of a goto process—, since we assume that there is no need to grant specific privileges in such circumstances.

The type system, described in figures 13–15, includes three kinds of judgements: (a) judgement $\Gamma \vdash \mathrm{env}$ asserts that Γ is a well-formed environment; (b) judgement $\Gamma \vdash_{s\tilde{t}} P : A$ means that process P is running at site s has travelled through the sequence of sites $\tilde{t}$, has the effects enumerated in set of actions A, and is well typed under typing assumptions Γ; and (c) judgement $\Gamma \vdash N$ denotes that network N is well typed under typing assumptions Γ.

Well-formed environment rules, figure 13, guarantee that group structures are not circular. Rule E-GROUP ensures that when we enlarge a typing with a new group definition, its parent groups are already in the typing. The remaining rules are simple.

As for processes (figure 14), rule P-OUTB enforces that typing Γ is well formed, and that channel a is a write or a read-write channel located at the site where the process is running and is capable of carrying unit values. Rule P-OUTC types an output process that carries another channel, rather than the unit value. The difference w.r.t P-OUTB regards the type for channel a: it must carry channels of the type of b and must be

$$\textsc{E-Unit}\quad \diamond\colon \mathsf{unit} \vdash \mathrm{env} \qquad \textsc{E-Channel}\quad \frac{\Gamma \vdash \mathrm{env}}{\Gamma, a\colon C@s \vdash \mathrm{env}}$$

$$\textsc{E-Site}\quad \frac{\Gamma \vdash \mathrm{env}}{\Gamma, s\colon G \vdash \mathrm{env}} \qquad \textsc{E-Group}\quad \frac{\Gamma \vdash \mathrm{env} \quad G \subseteq \mathrm{dom}(\Gamma)}{\Gamma, g\colon (\mathcal{P}, G) \vdash \mathrm{env}}$$

Figure 13: Well-formed environments.

$$\textsc{P-Outb}\quad \frac{\Gamma \vdash \mathrm{env} \quad \Gamma(a) <: \langle \mathsf{unit} \rangle\ @s}{\Gamma \vdash_{\widetilde{st}} a!\langle\diamond\rangle\colon \{\mathsf{useRes}\}} \qquad \textsc{P-Outc}\quad \frac{\Gamma \vdash \mathrm{env} \quad \Gamma(a) <: \langle C@G \rangle\ @s \quad \Gamma(r) = G \quad \Gamma(b) = C@r}{\Gamma \vdash_{\widetilde{st}} a!\langle b@r\rangle\colon \{\mathsf{useRes}\}}$$

$$\textsc{P-Inpb}\quad \frac{\Gamma \vdash_{\widetilde{st}} P\colon A \quad \Gamma(a) <: \langle \mathsf{unit} \rangle\ @s}{\Gamma \vdash_{\widetilde{st}} a?(\diamond)P\colon A \cup \{\mathsf{installRes}\}}$$

$$\textsc{P-Inpc}\quad \frac{\Gamma, x\colon C@y, y\colon G \vdash_{\widetilde{st}} P\colon A \quad \Gamma(a) <: \langle C@G \rangle\ @s \quad y \text{ not in } \Gamma}{\Gamma \vdash_{\widetilde{st}} a?(x@y)P\colon A \cup \{\mathsf{installRes}\}}$$

$$\textsc{P-Inpr}\quad \frac{\Gamma \vdash_{\widetilde{t}} a?(v)P\colon A}{\Gamma \vdash_{\widetilde{t}} a?*(v)P\colon A} \qquad \textsc{P-Par}\quad \frac{\Gamma \vdash_{\widetilde{t}} P\colon A_1 \quad \Gamma \vdash_{\widetilde{t}} Q\colon A_2}{\Gamma \vdash_{\widetilde{t}} P \mid Q\colon A_1 \cup A_2}$$

$$\textsc{P-Ress}\quad \frac{\Gamma, r\colon G@s \vdash_{\widetilde{st}} P\colon A \quad r \text{ not in } \Gamma}{\Gamma \vdash_{\widetilde{st}} (\boldsymbol{\nu} r\colon G)P\colon A \cup \{\mathsf{createSite}\}}$$

$$\textsc{P-Resc}\quad \frac{\Gamma, a\colon C@s \vdash_{\widetilde{st}} P\colon A}{\Gamma \vdash_{\widetilde{st}} (\boldsymbol{\nu} a\colon C)P\colon A \cup \{\mathsf{createRes}\}}$$

$$\textsc{P-Resg}\quad \frac{\Gamma, g\colon (\mathcal{P}, G)@s \vdash_{\widetilde{st}} P\colon A \quad g \text{ not in } \Gamma}{\Gamma \vdash_{\widetilde{st}} (\boldsymbol{\nu} g\colon (\mathcal{P}, G))\ P\colon A \cup \{\mathsf{createGroup}\}} \qquad \textsc{P-Nil}\quad \frac{\Gamma \vdash \mathrm{env}}{\Gamma \vdash_{\widetilde{t}} \mathsf{stop}\colon \emptyset}$$

$$\textsc{P-Mig}\quad \frac{\Gamma \vdash_{r\widetilde{t}} P\colon A \quad \Gamma \vdash r \text{ allows } \mathcal{P}\colon A \quad \Gamma \vdash \mathcal{P} \text{ canEnter } r}{\Gamma \vdash_{\widetilde{t}} \mathsf{goto}\ r.P\colon \emptyset}$$

Figure 14: Typing processes.

located at the groups of site r. To type an input process, $a?(x@y)P$, channel a must be a read or a read-write channel. The continuation process, P, must be well typed in a typing augmented with x and y. Notice that channel x is located at y and that y is defined as a site member of the groups that channel a can carry. Hence, we guarantee that the privileges for the actions involving x and y are correctly checked, since we verify policies against all groups in G. The subtyping rule is covariant for inputs, which means that, if the type of channel a is a subtype of $\langle C@G \rangle\ @s$, then a carries channels located at a subset of G. Finally, the effect of the input action—installRes—is appended to the set of actions. Rules P-Inpb and P-Inpr follow a pattern similar

$$\dfrac{\Gamma \vdash_s P : A}{\Gamma \vdash s[P]}\ \text{(N-SITE)} \qquad \dfrac{\Gamma, n : L@s \vdash N \quad n \text{ not in } \Gamma}{\Gamma \vdash (\nu n : (L, @)s)\, N}\ \text{(N-RES)}$$

$$\dfrac{\Gamma \vdash N \quad \Gamma \vdash M}{\Gamma \vdash N \,|\, M}\ \text{(N-PAR)} \qquad \dfrac{\Gamma \vdash \mathsf{env}}{\Gamma \vdash \mathsf{stop}}\ \text{(N-NIL)}$$

Figure 15: Typing networks.

$$\dfrac{\Delta \vdash_{\widetilde{st}} P : A \quad \Delta \vdash s \ \mathsf{allows}\ \mathbf{e} : A \qquad \Gamma <: \Delta \quad \Delta \vdash \mathbf{e}\ \mathsf{canEnter}\ s}{\Gamma \Vdash s[P]^{\tilde{t}}_{\Delta}}\ \text{(T-SITE)} \qquad \dfrac{\Gamma, a : C@s \Vdash N \qquad \Gamma \vdash s \ \mathsf{allows}\ \mathbf{e} : \ \mathsf{createRes}}{\Gamma \Vdash (\nu_{\tilde{t}} a : C@s)\, N}\ \text{(T-RESC)}$$

$$\dfrac{\Gamma, s : G@s \Vdash N \qquad \Gamma \vdash s \ \mathsf{allows}\ \mathbf{e} : \ \mathsf{createSite}}{\Gamma \Vdash (\nu_{\tilde{t}} s : G)\, N}\ \text{(T-RESS)} \qquad \dfrac{\Gamma, g : (\mathcal{P}, G)@s \Vdash N \quad g \text{ not in } \Gamma \qquad \Gamma \vdash s \ \mathsf{allows}\ \mathbf{e} : \ \mathsf{createGroup}}{\Gamma \Vdash (\nu_{\tilde{t}} g : (\mathcal{P}, G))\, N}\ \text{(T-RESG)}$$

(*plus the rules in figure 15, except* N-SITE, *and* N-RES)

Figure 16: The tagged language – typing networks.

to the one just described.

The parallel composition of processes, $P \,|\, Q$, combines the set of actions gathered when typing the individual processes. We split name restriction over three rules, P-RESS, P-RESC, and P-RESG, since there is a specific effect associated to each creation action.

It is at code migration, goto $r.P$, that all the security checking takes place. When we reach a goto process we have all the information necessary to check security polices, namely, we know the sequence of sites visited by code (annotated under the turnstile), the target site (indicated in the syntax of the goto process), as well as the actions performed by process P (the set A in the typing for P) that need to be checked in the typing. Therefore, the typing of a goto process checks if the continuation process is well typed and ensures that the target site allows code that travels through that sequence of sites to perform the actions of the continuation process. There is no effect associated with code migration, since the actions that P performs are checked at this level, and so, there is no point in including the actions executed by P to be checked again at outermost levels.

Figure 15 describes network typing. Security policies are not checked at network level, since no computation takes place at this level. Therefore, the only interesting fact to stress (refer to rule N-SITE) is that when code is installed at a certain site, the actions that are not the continuation of a goto process are not checked. Indeed, since functions allows and canEnter are reflexive, there is no point in checking policies at this level.

The type system we present preserves typings during reduction.

Theorem 2.1 (Subject Reduction). *If* $\Gamma \vdash N$ *and* $N \to M$, *then* $\Gamma \vdash M$.

Typing the tagged language. The changes imposed on the type system by tagging are described in figure 16. The typing rules for processes are left unchanged, but at network level we propose substantially different rules. Rule T-SITE checks, among other things, that the tagged typing assumptions are enough to type the process. Moreover, the conclusion's typing environment, Γ, is a relaxed version of the one used for tagging,

Δ. Hence, it is possible to consider the minimum security requirements to type a process, and then enlarge the set of security properties at network level.

Notice that we now verify security policies in the rule, since the inclusion of the path travelled by the code ($\mathbf{e}$) represents an implicit goto process. To understand the need to check security policies at site level, consider, for instance, sequent $\Delta \vdash_{s\tilde{t}} P : A$. It is always possible to run a process at its host site: $\Gamma \Vdash s[P]^s_\Delta$, for $\Gamma <: \Delta$. But if we want to indicate that the code migrated from a site, say t, $s[P]^t_\Delta$, then it is only possible if site s allows t to execute actions in set A and t is able to enter s border.

We check also the declaration of new names, since we are again in presence of an implicit goto. From the network declaration $(\boldsymbol{\nu}_{\tilde{t}} n : L_{@s})\, N$ we infer that there was a code migration all along the sequence of sites $\tilde{t}$ to site s, and then afterwards the creation of n took place. The remaining typing rules result from syntax modifications.

Theorem 2.2 (Type preservation). *If $\Gamma \Vdash N$ and $N \mapsto M$, then $\Gamma \Vdash M$.*

Type safety. The tagged and untagged reduction relations are closely related. We define a tag function $\mathsf{tag}_\Gamma(N)$ in [41] that takes an untagged network N and yields the set of tagged networks obtained from it using Γ.

The following results ensure that types are preserved both by the tagging function and by the tagged reduction.

Theorem 2.3 (Type preservation). *If $\Gamma \vdash N$, and $M \in \mathsf{tag}_\Gamma(N)$, then $\Gamma \Vdash M$.*

Theorem 2.4 (Operational correspondence).

1. *If $N \to N'$, then $\exists M \in \mathsf{tag}_\Gamma(N)$ s.t. $M \mapsto M' \in \mathsf{tag}_\Gamma(N')$.*
2. *If $M \mapsto M'$, then $\exists N \in \mathsf{tag}_\Gamma(N), N' \in \mathsf{tag}_\Gamma(M')$ s.t. $N \to N'$.*

The type safety result states that well-typed networks do not incur in runtime errors.

Theorem 2.5 (Type safety). *If $\Gamma \Vdash M$, then $M \not\xrightarrow{err}$.*

2.4 Concluding Remarks

Conclusion. We present an approach to express and control history-based access to resources using types. We use Dπ as the underlying calculus and, on top of it, define a hierarchical structure of security groups. The security model we propose is based on the notion of security group that delimits a region of the network with the same security requirements. Security groups may be understood as a firewall that dictates and supervises the sites under its control. We use a type system as the security mechanism to enforce that networks respect the security policies defined by groups and claim a type safety result.

Ongoing work comprise the refinement of the type system to enforce a fine grained control of resources' security and the study of how to change policies dynamically.

Related work. Refer to [3] for a general survey on concurrent mobile calculi, type systems, and security policies. As far as we known, our security model is the first to mix group policies, to record history-based access to resources, and to use a group hierarchy for helping the writing of security rules and the reusing of existent ones.

Cardelli, Ghelli, and Gordon introduced the notion of groups for the Ambient calculus [10, 12, 14] to control the movement and the opening of ambients. They use groups to combine ambients in clusters, but specify the security properties for each ambient regardless the group the ambient belongs to. Instead, we use groups to specify security policies shared by the sites that compose each group.

Lhoussaine and Sassone [39] use dependent types as an alternative to groups. The type system is far more complex and the calculus does not facilitates the writing of policies.

The work on Dπ has proposed advanced type systems [32, 34] to control resource access. The control of policies is based on a subtype relation that permits the delivery of different types of the same channel to distinguished parties. Code mobility is controlled with the **go** keyword. If a

As a matter of fact, the open primitive of MA has been considered by many researchers as potentially dangerous, because it could be inadvertently or maliciously used to destroy an ambient's individuality (by dissolving its boundary). Several variants of MA have therefore been proposed, which either are equipped with additional constructs for controlling the execution of open, like co-capabilities of Safe Ambients [37], or replace it with other mechanisms for interaction between ambients, such as the communication between nested ambients that characterizes Boxed Ambients [6]. Process mobility, however, seems to be a more suitable mechanism for modelling code exchange and remote execution.

The rest of this section is organized as follows. We first describe the main features of our system; then we apply it to a meaningful example, which illustrates the use of the different constructs of the calculus. Finally, we draw some short conclusions. Most technical details, like the proof of subject reduction, have been omitted. We refer the interested reader to the full paper [16] for a detailed account. In it we also introduce, along with further examples, a behavioural semantics (i.e., barbed congruence) and some equivalence laws based on it. The soundness of the laws is proved through standard techniques by relying on a higher-order labelled transition system and on a labelled bisimulation (which is proved sound w.r.t. to the behavioural semantics).

3.1 The typed language and its reduction semantics

The syntax of the pre-terms of the language (where type constraints are ignored) is given in Fig. 17, the one of types in Fig. 18. The precise syntax of the language results from the typing rules given in Definition 19. Processes are built through the usual constructs of sequential action prefixing, parallel composition, and ambient construction. In the following a process of the form $m{:}g(G)[\mathsf{c},\mathsf{e}\|P]$, which corresponds, in our richer calculus, to the term $m[P]$ of standard ambient calculi, will be simply called an ambient, distinct from the mere ambient name m. Communication is only local (and synchronous), as in the original MA [14]. Actions include the usual in and out primitives for ambient mobility, and the two new primitives down and up for moving processes between ambients; already taken into consideration in [17], they replace the to primitive of $\mathbf{M}^3$.

The down action is analogous to the in action. Its (simplified) reduction rule (see [17]) is: $\mathsf{down}\, m.P \mid Q \mid m[R] \rightarrow Q \mid m[P \mid R]$, i.e., the process $\mathsf{down}\, m.P$ enters an ambient named m where it continues as P. On the contrary, the up action is only partially analogous to out, since its argument is the name of the destination ambient and not that of the source ambient, like in the case of out. The corresponding (simplified) reduction rule is $m[n[\mathsf{up}\, m.P \mid R] \| Q] \rightarrow m[P \mid n[R] \| Q]$, also given in [17]. The explicit mention of the target ambient allows more effective controls on the incoming process. In the present setting the four mobility primitives, though keeping their basic behaviours just recalled, come with richer syntactic forms that correspond to more sophisticated reduction rules, as we will see later in the section.

Types describe communication and mobility properties. With reference to an ambient, we distinguish between its active and passive mobility: by the former we intend which ambients the given ambient may cross or send processes to; by the latter, the ambients by which it may be crossed or sent processes. Of course, by directly specifying the active mobility of each ambient one indirectly specifies the passive mobility of all the concerned ambients, and vice versa. One of the main features of our system is that static types directly specify active mobility while dynamic types directly specify passive mobility, and the compatibility between them is tested by runtime checks.

The type system is based on *ambient groups*: a group is a name that represents (i.e., labels) a set of ambient name occurrences. Different ambient names may belong to the same group, but at the same time different occurrences of the same ambient name may be labelled with different group names, i.e., different ambients with the same name may belong to different groups. The mobility properties directly specified for each ambient are always expressed with reference to groups and not to individual ambients, so as to avoid a dependence of types from values.

In order to enable an ambient to check at runtime (i.e., during reduction) that the active types of the

A denotes the set of *ambient names* and G denotes the set of *group names*.

$\alpha ::=$ **ambient name**

$m, n, \ldots$	ambient names
$x, y, \ldots$	variables

$\gamma ::=$ **group name**

$g, h, \ldots$	group names
$x, y, \ldots$	variables

$\chi ::=$ **capability**

$\mathsf{in}\,\alpha{:}g$	moves the containing ambient into ambient α of group g
$\mathsf{out}\,\alpha{:}g$	moves the containing ambient out of ambient α of group g
$\mathsf{up}\,\alpha{:}g\,\mathsf{with}\,G$	moves the continuation process out from its ambient up to enclosing ambient α of group g requiring rights G
$\mathsf{down}\,\alpha{:}g\,\mathsf{with}\,G$	moves the continuation process from its ambient down to enclosed ambient α of group g requiring rights G
$\mathsf{add}\ \gamma^{\varphi}\ \mathsf{in}\ \alpha{:}g$	adds the crossing right γ with multiplicity φ to ambient α of group g
$\mathsf{add}^{\sim}\ \langle\gamma, G\rangle^{\varphi}\ \mathsf{in}\ \alpha{:}g$	adds the entering right $\langle\gamma, G\rangle$ with multiplicity φ to ambient α of group g
$\chi.\chi'$	path
$x, y, \ldots$	variables

$M, N, L ::=$ **general**

α	ambients
γ	groups
χ	capabilities

$\rho ::=$ **guard**

χ	capabilities
$\langle\overrightarrow{M}\rangle$	synchronous output
$(\overrightarrow{x{:}W})$	typed input

$P, Q, R ::=$ **process**

0	null
$\rho.P$	prefixing
$P \mid Q$	parallel composition
$\alpha{:}g(G[\mathsf{c}, \mathsf{e}] \mid P]$	ambient
$!\rho.P$	guarded replication
$(\nu n)P$	name restriction

where c are multisets of groups, e are multisets of pairs $\langle g, G\rangle$; $\overrightarrow{W}$ and G are defined in Figure 18.

Figure 17: Syntax

		$\mathsf{C}, \mathsf{E}, \ldots$	sets of groups;
G	::=	$\mathsf{mc}(\mathsf{C}, \mathsf{E}, T)$	**mobcom** type: mobility rights and communication type
Proc	::=	$g(G)$	process type: processes that can stay in ambients of group g with rights G
Cap	::=	$g'(G')\ \ g(G)$	capabilities that can be consumed by processes of type $g'(G')$ and leave processes of type $g(G)$ as continuations
W	::=		message type
		Cap	capability type
		group	group
		amb	ambient type
T	::=		communication type
		shh	no communication
		$\overrightarrow{W}$	communication of messages of type $\overrightarrow{W}$
Σ	::=	$\varnothing$	context
		$\Sigma, x : W$	

Figure 18: Types

incoming processes are compatible with its own type, the primitives for moving processes between ambients carry with them the communication and mobility types of their continuations.

While local typing allows the control of the active mobility behaviour, the absence of global type information makes impossible a static control of the passive behaviour. This check has therefore also to be performed dynamically; for that reason, at runtime each ambient carries a specification of which (groups of) ambients can cross it and which can send it processes, and how many times.

Thus every mobility action becomes subject to a double authorization, one static and the other dynamic. The fact that "passive" permits are dynamically checked allows them to also be dynamically granted; to this end, we have introduced two new primitives through which a process may enrich the rights of another one thus enabling it to carry out a given task.

3.1.1 Static typing for mobility

The static type system is centered on the notion of process type, which consists of a group name g and of a mobility and communication type (or **mobcom** type for short) G; following the notation of [10], we write it $g(G)$.

The mobcom type G is of the form $\mathsf{mc}(\mathsf{C}, \mathsf{E}, T)$, where C is the set of (groups of) ambients into which the process may drive (through an in or out action) its enclosing ambient, E the set of (groups of) ambients to which it may send (through a down or up action) a continuation process, and T is the process communication type. We use the notation $\mathsf{C}(G)$, $\mathsf{E}(G)$, $T(G)$ to respectively indicate the components C, E, T of G.

Like in most ambient calculi, all the (parallel) processes within an ambient must have the same process type $g(G)$, which is thus a sort of inner type of that ambient and, as we will see, is bound locally to it. Ambient names, on the other hand, only have the atomic type amb, and are therefore omitted in the environments. As a consequence, name restriction may be simply written as $(\nu m)P$. Similarly, group names have the atomic type group. Group names can be exchanged in communications but group variables can only be used in a limited way, as will be remarked later.

Ambient and process mobility actions must specify not only an ambient name, as in MA, but also a

group name. For instance, the syntax of the usual in and out primitives becomes in α:g, out α:g (with α ambient name or variable). A process may contain one such action (i.e., it may be well typed) only if its type allows it to drive its enclosing ambient across the boundary of ambients labelled with the group name g. Values exchanged in communications may be ambient names (of type amb), group names (of type group), or capabilities.

A capability type consists, as in [17], of a pair of process types, written here with the notation $g'(G')$ $g(G)$. Capability types take into account the fact that, owing to the down or up actions, a process can move from one ambient to another, changing its type accordingly. A capability χ of type $g'(G')$ $g(G)$ drives a continuation process P from an ambient of (inner) type $g'(G')$ to an ambient of type $g(G)$. Obviously P must have the type $g(G)$ of the destination ambient while the resulting process $\chi.P$ has the type $g'(G')$ of the source ambient. This is formalized in the rule (PREFIX) of Fig. 19. The type of a sequence of in- or out-capabilities has the form $g(G)$ $g(G)$, because in executing in or out actions a process remains in the same ambient.

The meaning of types, informally described in the body of this section, is formally defined by the set of the typing rules shown in Fig. 19.

3.1.2 [illegible]

Type controls no longer statically performed must of course be done dynamically. To this end, the inner type $g(G)$ of an ambient named m is bound to it with the notation m:$g(G[\ldots]$ An ambient is also characterized by two components c and e that record by which ambient or process groups it may be entered and how many times. The complete notation is α:$g(G[$c, e$\|P]$ with α variable or ambient name.

More precisely, in a given ambient the component c is the multiset of groups of ambients that are allowed to cross its external boundary, while e is the multiset of groups of ambients that are allowed to send processes to it. In e each element is actually a pair $\langle g, G\rangle$ consisting of a group and a mobcom type; its meaning is that a process coming from an ambient of group g is given the entrance permit if it respects the behavioural constraints specified by the type G. In an ambient m:$g_m(G_m[$c, e$\|P]$ all the pairs $\langle g, G\rangle$ occurring in e must therefore be such that G is a subtype of G_m.

Each execution of an in or out action consumes one element of c and each execution of a down or up action consumes one element of e, with the exception of [illegible] elements, which represent permanent permits, i.e., elements with infinite multiplicity. The control of the mobility constraints represented by c and e is performed dynamically during process reduction. A reduction rule cannot fire if the corresponding side conditions on c and e are not satisfied.

In the following we give the formal definition of a multiset with possibly infinite multiplicities, and we define the operations of addition and removal of elements.

[illegible] 3.1

- A [illegible] over a set S is a function from S to the set $\omega \cup \{*\}$ (where ω is the set of natural numbers and $*$ is the infinite cardinality).
- If f is a [illegible] on S, $s, r \in S$, $\varphi \in \omega \cup \{*\}$ we define
  - $s \in f$ if $f(s) \neq 0$;
  - $(f \cup s^{\varphi})(r) = \begin{cases} f(s) + \varphi & \text{if } r = s, \\ f(r) & \text{otherwise.} \end{cases}$
  - $(f \downarrow s)(r) = \begin{cases} f(s) - 1 & \text{if } r = s, \\ f(r) & \text{otherwise.} \end{cases}$

  where $e * \pm \varphi = *$.

$$\frac{n \in \mathsf{A}}{\Theta;\Sigma \vdash n : \mathsf{amb}}\ (\textsc{Amb Const}) \qquad \frac{g \in \mathsf{G}}{\Theta;\Sigma \vdash g : \mathsf{group}}\ (\textsc{Grp Const})$$

$$\frac{x : W \in \Sigma}{\Theta;\Sigma \vdash x : W}\ (\textsc{Env}) \qquad \frac{}{\Theta;\Sigma \vdash 0 : ((g)G)}\ (\textsc{Null})$$

$$\frac{\Theta;\Sigma \vdash \alpha : \mathsf{amb} \quad g \in \mathsf{C}(G)}{\Theta;\Sigma \vdash \mathsf{in}\,\alpha{:}g : g^{0}(G) \quad g^{0}(G)}\ (\textsc{In})$$

$$\frac{\Theta;\Sigma \vdash \alpha : \mathsf{amb} \quad g \in \mathsf{E}(G^{0})}{\Theta;\Sigma \vdash \mathsf{down}\,\alpha{:}g\,\mathsf{with}\,G : g^{0}(G^{0}) \quad g(G)}\ (\textsc{Down})$$

$$\frac{\Theta;\Sigma \vdash \alpha : \mathsf{amb} \quad g \in \mathsf{C}(G)}{\Theta;\Sigma \vdash \mathsf{out}\,\alpha{:}g : g^{0}(G) \quad g^{0}(G)}\ (\textsc{Out}) \qquad \frac{\Theta;\Sigma \vdash \alpha : \mathsf{amb} \quad g \in \mathsf{E}(G^{0})}{\Theta;\Sigma \vdash \mathsf{up}\,\alpha{:}g\,\mathsf{with}\,G : g^{0}(G^{0}) \quad g(G)}\ (\textsc{Up})$$

$$\frac{\langle g; g^{0}\rangle \in \Theta \quad \Theta;\Sigma \vdash \alpha : \mathsf{amb} \quad \Theta;\Sigma \vdash \gamma : \mathsf{group}}{\Theta;\Sigma \vdash \mathsf{add}^{c}\,\gamma^{\varphi}\,\mathsf{in}\,\alpha{:}g : g^{0}(G) \quad g^{0}(G)}\ (\textsc{Add-c})$$

$$\frac{\langle g; g^{0}\rangle \in \Theta \quad \Theta;\Sigma \vdash \alpha : \mathsf{amb} \quad \Theta;\Sigma \vdash \gamma : \mathsf{group}}{\Theta;\Sigma \vdash \mathsf{add}^{e}\,\langle\gamma; G\rangle^{\varphi}\,\mathsf{in}\,\alpha{:}g : g^{0}(G^{0}) \quad g^{0}(G^{0})}\ (\textsc{Add-e})$$

$$\frac{\Theta;\Sigma \vdash \chi : g^{0}(G^{0}) \quad g^{0}(G^{0}) \quad \Theta;\Sigma \vdash \chi^{0} : g^{0}(G^{0}) \quad g(G)}{\Theta;\Sigma \vdash \chi{:}\chi^{0} : g^{0}(G^{0}) \quad g(G)}\ (\textsc{Path})$$

$$\frac{\Theta;\Sigma \vdash \chi : g^{0}(G^{0}) \quad g(G) \quad \Theta;\Sigma \vdash P : g(G)}{\Theta;\Sigma \vdash \chi{:}P : g^{0}(G^{0})}\ (\textsc{Prefix})$$

$$\frac{\Theta;\Sigma;x : \overset{!}{W} \vdash P : g(G) \quad G \quad \mathsf{mc}(\mathsf{C};\mathsf{E};\overset{!}{W})}{\Theta;\Sigma \vdash (x : \overset{!}{W}) : P : g(G)}\ (\textsc{Input})$$

$$\frac{\Theta;\Sigma \vdash P : g(G) \quad \Theta;\Sigma \vdash \overset{!}{M} : \overset{!}{W} \quad G \quad \mathsf{mc}(\mathsf{C};\mathsf{E};\overset{!}{W})}{\Theta;\Sigma \vdash \langle\overset{!}{M}\rangle : P : g(G)}\ (\textsc{Output})$$

$$\frac{\Theta;\Sigma \vdash \alpha : \mathsf{amb} \quad \Theta;\Sigma \vdash P : g(G) \quad \langle g^{0}; G^{0}\rangle \in \mathsf{e} = \Theta \quad G^{0} \quad G}{\Theta;\Sigma \vdash \alpha{:}g(G)[\mathsf{c};\mathsf{e}\,k P] : g^{0}(G^{0})}\ (\textsc{Amb})$$

$$\frac{\Theta;\Sigma \vdash P : g(G) \quad \Theta;\Sigma \vdash Q : g(G)}{\Theta;\Sigma \vdash P \mid Q : g(G)}\ (\textsc{Par}) \qquad \frac{\Theta;\Sigma \vdash \rho : P : g(G)}{\Theta;\Sigma \vdash\, !\,\rho : P : g(G)}\ (\textsc{Repl})$$

$$\frac{\Theta;\Sigma \vdash P : g(G)}{\Theta;\Sigma \vdash (\nu n)P : g(G)}\ (\textsc{Amb Res})$$

Figure 19: Typing rules

A partial order on mobcom types is naturally defined via set inclusion, and so is the notion of glb of mobcom types. Communication subtyping is characterized only by the fact that shh is smaller than any other communication type.

Definition 3.2

- $T \leq T'$ if and only if $T = \mathsf{shh}$ or $T = T'$.
- $\mathsf{mc}(\mathsf{C},\mathsf{E},T) \leq \mathsf{mc}(\mathsf{C}',\mathsf{E}',T')$ if $\mathsf{C} \subseteq \mathsf{C}'$ and $\mathsf{E} \subseteq \mathsf{E}'$ and $T \leq T'$.
- $T \sqcap T' = \begin{cases} \mathsf{shh} & \text{if } T = \mathsf{shh} \text{ or } T' = \mathsf{shh} \\ T & \text{if } T = T' \\ \text{undefined} & \text{otherwise} \end{cases}$
- $\mathsf{mc}(\mathsf{C},\mathsf{E},T) \sqcap \mathsf{mc}(\mathsf{C}',\mathsf{E}',T') = \mathsf{mc}(\mathsf{C} \cap \mathsf{C}', \mathsf{E} \cap \mathsf{E}', T \sqcap T')$ if $T \sqcap T'$ is defined and undefined otherwise.

The elements of c and e are similar to the co-capabilities of Safe Ambients [37], with starred elements corresponding to banged co-capabilities.

3.1.3 Dynamic modification of boundaries

The components c and e of an ambient process may allow or forbid movements at runtime; they can therefore be changed dynamically without breaking the subject reduction. As a matter of fact, this is achieved:

- by automatically removing a (consumable) permit when a movement action is performed;
- by adding to c or to e an element with a multiplicity, by means of one of the two newly introduced permit-adding primitives add and $\widetilde{\text{add}}$.

Action $\text{add } g^{\varphi} \text{ in } m{:}g_m$ dynamically adds the group g with multiplicity φ to the c component of a local ambient named m of group g_m (see the reduction rule (R-add)); as usual, by "local ambient" we intend one that is found in the same enclosing ambient. Action $\widetilde{\text{add}}\ \langle g, G_1\rangle^{\varphi} \text{ in } m{:}g_m$ dynamically adds the group/type pair $\langle g, G_1\rangle$ with multiplicity φ to the e component of a local ambient m (rule (R-$\widetilde{\text{add}}$)). In a term $m{:}g_m(G_m[\mathsf{c},\mathsf{e}\|P])$ all mobcom types occurring in e are subtypes of G_m. This property is preserved by the reduction rule (R-$\widetilde{\text{add}}$) since the $\widetilde{\text{add}}\ \langle g, G_1\rangle^{\varphi} \text{ in } m{:}g_m$ action can be performed only if $G_1 \sqcap G_m$ is defined.

A process may perform a permit-adding operation on an ambient only if its group is higher than the target's group in a global administrative hierarchy, represented by a partial order relation $\sqsubset$ over group names. Such hierarchy is the only global environment of our calculus; it might be thought of as some general (not necessarily centralized) coordination and administration structure of the network. The typing rules (ADD-C), (ADD-E) assure that this hierarchy is always respected.

3.1.4 Mobility and migrating processes

Process mobility actions must specify, in addition to the ambient and group name of the destination, the mobcom type G of the continuation process (i.e., of the process that will run within the target ambient). The complete syntax of the down action is $\text{down } m{:}g_m \text{ with } G$, that of up is similar. Of course, the type G needs to be compatible (via subtyping) with the mobcom type G_m of the destination ambient. More precisely, if a process $\text{down } m{:}g_m \text{ with } G'.P$ is of group g (i.e., is typed with a type $g(G)$) the e component of the target ambient m must contain the group g paired with a type G'' such that $G' \leq G''$. The typing rule (AMB), along with the reduction rule (R-$\widetilde{\text{add}}$), ensures that $G'' \leq G_m$ and so $G' \leq G'' \leq G_m$. Hence, the migrating

Reduction rules

$$\begin{array}{ll}
\text{(R-in)} & n[g_n(G_n)[\mathsf{c}_n, \mathsf{e}_n \| \text{in } m.P \mid Q] \mid m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \| R] \\
[\beta pt] & \quad \to_{g;G} \; m[g_m(G_m)[\mathsf{c}_m \downarrow g_n, \mathsf{e}_m \| n[g_n(G_n)[\mathsf{c}_n, \mathsf{e}_n \| P \mid Q] \| R] \\
 & \quad \text{if } g_n \in \mathsf{c}_m \\
\text{(R-out)} & m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \| n[g_n(G_n)[\mathsf{c}_n, \mathsf{e}_n \| \text{out } m.P \mid Q] \| R] \\
[\beta pt] & \quad \to_{g;G} \; n[g_n(G_n)[\mathsf{c}_n, \mathsf{e}_n \| P \mid Q] \mid m[g_m(G_m)[\mathsf{c}_m \downarrow g_n, \mathsf{e}_m \| R] \\
 & \quad \text{if } g_n \in \mathsf{c}_m \\
\text{(R-down)} & \text{down } m[g_m] \text{ with } G'.P \mid m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \| Q] \\
 & \quad \to_{g;G} \; m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \downarrow \langle g, G'' \rangle \| P \mid Q] \\
 & \quad \text{if } \langle g, G'' \rangle \in \mathsf{e}_m \,\&\, G' \leq G'' \\
\text{(R-up)} & m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \| n[g_n(G_n)[\mathsf{c}_n, \mathsf{e}_n \| \text{up } m[g_m] \text{ with } G'.P \mid R]] \mid Q] \\
 & \quad \to_{g;G} \; m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \downarrow \langle g_n, G'' \rangle \| n[g_n(G_n)[\mathsf{c}_n, \mathsf{e}_n \| R]] \mid Q \mid P] \\
 & \quad \text{if } \langle g_n, G'' \rangle \in \mathsf{e}_m \,\&\, G' \leq G'' \\
\text{(R-add}^{\wedge}\text{)} & \text{add}^{\wedge} \; g'^{\varphi} \text{ in } m[g_m].P \mid m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \| R] \\
 & \quad \to_{g;G} \; P \mid m[g_m(G_m)[\mathsf{c}_m \cup g'^{\varphi}, \mathsf{e}_m \| R] \\
\text{(R-add}^{\sim}\text{)} & \text{add}^{\sim} \; \langle g', G' \rangle^{\varphi} \text{ in } m[g_m].P \mid m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \| R] \\
 & \quad \to_{g;G} \; P \mid m[g_m(G_m)[\mathsf{c}_m, \mathsf{e}_m \cup \langle g', G' \sqcap G_m \rangle^{\varphi} \| R] \\
 & \quad \text{if } G' \sqcap G_m \text{ is defined} \\
\text{(R-comm)} & (\overrightarrow{x\colon W}).P \mid \langle \vec{M} \rangle.Q \;\to_{g;G}\; P\{\vec{x} := \vec{M}\} \mid Q
\end{array}$$

Reduction rules

$$\begin{array}{lll}
\text{(R-par)} & P \to_{g;G} Q & \Rightarrow \; P \mid R \to_{g;G} Q \mid R \\
\text{(R-amb)} & P \to_{g;G} Q & \Rightarrow \; n[g(G)[\mathsf{c}, \mathsf{e} \| P]] \to_{g^{\emptyset};G^{\emptyset}} n[g(G)[\mathsf{c}, \mathsf{e} \| Q]] \\
\text{(R-}\equiv\text{)} & P' \equiv P', \; P \to_{g;G} Q, \; Q \equiv Q' & \Rightarrow \; P' \to_{g;G} Q' \\
\text{(R-}\nu\text{)} & P \to_{g;G} Q & \Rightarrow \; (\nu n)P \to_{g;G} (\nu n)Q
\end{array}$$

Structural congruence $\equiv$: $(\mid, 0)$ is a commutative monoid.

$$(\nu n)\, P \mid Q \equiv (\nu n)P \mid Q \quad (n \notin \text{fn}(Q)) \qquad (\nu n)\,(\nu m)P \equiv (\nu m)\,(\nu n)P$$

$$n[g(G)[\mathsf{c}, \mathsf{e} \| (\nu m)P]] \equiv (\nu m)n[g(G)[\mathsf{c}, \mathsf{e} \| P]] \quad (n \neq m) \qquad !P \equiv P \mid !P$$

Figure 20: Reduction

process P is guaranteed not to require more rights than those specified by the inner type $g_m(G_m)$ of the destination ambient, which was statically checked.

Reduction rules are thus dependent on the typing assumptions and the reduction relation is labelled with the process type $g(G)$, even though the group name g is only involved in the (R-down) rule and the type G never plays any role in reduction. In fact reduction rules are only defined for well typed terms, i.e., for

processes that are typed with some type $g(G)$. The complete set of rules is given in Fig. 20.

A basic property of the system is that typing is preserved by $\leq$ on mobcom types.

Lemma 3.3 *If $\mathcal{O};\Sigma \vdash P : g(G)$ and $G \leq G'$ then $\mathcal{O};\Sigma \vdash P : g(G')$*

Using Lemma 3.3 a property of subject reduction, which ensures that static typing is preserved by computation, can be proved with standard techniques.

Theorem 3.4 (Subject Reduction) *If $\mathcal{O};\Sigma \vdash P : g(G)$ and $P \rightarrow_{g;G} Q$ then $\mathcal{O};\Sigma \vdash Q : g(G)$.*

Finally, observe that group variables may only occur as first arguments of add and add$^{\sim}$, so that they never occur in G or within the c and e components, since otherwise their role in allowing a safe name restriction would be defeated.

3.2 An example

Our main example is the modelling of a public transportation system, the *train* introduced by [9] as a nice pictorial illustration of the issues related to the control of mobility.

We want to represent a railway network connecting a set of different places (e.g., cities) in the world. Trains move between stations, travellers may get into and off trains only at stations and cannot drive them (no hijacking possible). The number of passengers in a train at any given instant cannot exceed the number of seats; a passenger takes a seat on boarding and releases it on getting off. Each train has a fixed route.

For the sake of simplicity, we assume that:

- There is a top-level untrusted ambient *world*, which includes stations, travellers, and some other unspecified process R (e.g., other means of transport); it has group and mobcom type $g_w(G_w)$, but no assumption can be made on G_w. Also, c_{world}, e_{world} and R are unknown.
- In our intended representation different stations should be found within different cities or localities, and moving from one city to another would only be possible by train. The presence of cities would however increase the size of the example in a trivial manner, without providing more insights; we therefore place stations directly within *world*, although in this way travellers appear to use a train to end up in the same ambient *world* where they started from.
- There are only two stations A and B, and one train TRAIN commuting between them. Initially, the train is within A.

Stations and trains are represented by ambient processes; travellers are represented by simple processes; the number of free seats in a train is represented by the multiplicity of the right to get into the train.

Stations are ambients $A : g_{st}(G_{st})[\ldots]$ and $B : g_{st}(G_{st})[\ldots]$ of group g_{st} and mobcom type G_{st}. They are immobile, and can have travellers both going down into the trains or up into the world; they can be crossed by trains, and can receive travellers both from train and from the outside world. Correspondingly:

$$G_{st} = \mathsf{mc}(\varnothing, \{g_{tr}, g_w\}, \mathsf{shh}) \quad \mathsf{c}_{st} = \{g_{tr}^*\} \quad \mathsf{e}_{st} = \{\langle g_{tr}, G_{to}\rangle^*, \langle g_w, G_{from}\rangle^*\}$$

where $G_{from} = \mathsf{mc}(\varnothing, \{g_{tr}\}, \mathsf{shh})$ and $G_{to} = \mathsf{mc}(\varnothing, \{g_w\}, \mathsf{shh})$. Note that $G_{from} \leq G_{st}$ and $G_{to} \leq G_{st}$, i.e., both G_{from} and G_{to}, which represent two accepted behaviours for processes entering the station, are compatible with G_{st}, as is required by the typing rule (AMB).

The train is an ambient of group g_{tr}, which can cross stations and world, send traveller processes into stations, and receive a maximum number n of passengers from stations, provided they behave as good passengers (and not, for example, as drivers):

$$\text{TRAIN} \quad tr : g_{tr}(G_{tr})[\,!\mathsf{c}_{tr}, \mathsf{e}_{tr} \,\|\, !\mathsf{out}_A : g_{st}.\mathsf{in}_B : g_{st}.\mathsf{out}_B : g_{st}.\mathsf{in}_A : g_{st}\,]$$
$$\text{where } G_{tr} = \mathsf{mc}(\{g_{st}\}, \{g_{st}\}, \mathsf{shh}) \quad \mathsf{c}_{tr} = \varnothing \quad \mathsf{e}_{tr} = \{\langle g_{st}, G_{psng}\rangle^{\mathsf{n}}\}$$
$$\text{with } G_{psng} = \mathsf{mc}(\varnothing, \{g_{st}\}, \mathsf{shh})$$

A traveller is represented by a parametric process $\text{TRAVELLER}(s, c, d)$ which from some unspecified place in the world enters the station s c to become a passenger of a train that takes it to the station d :

$$\begin{array}{l}\text{TRAVELLER}(s, c, d) \triangleq \mathsf{down}\, s\, c{:}g_{st}\, \mathsf{with}\, G_{from}\, .\, \text{PSNG} \\ \text{where PSNG} \triangleq \mathsf{down}\, t : g_{tr}\, \mathsf{with}\, G_{psng}\, .\, \mathsf{up}\, d : g_{st}\, \mathsf{with}\, G_{to} \\ \qquad .\, \mathsf{add}^{\sim} \langle g_{st}, G_{psng} \rangle\, \mathsf{in}\, t : g_{tr}\, .\, \mathsf{up}\, w : g_w\, \mathsf{with}\, G_w\, .\, \text{P}\end{array}$$

The mobcom types G_{from} and G_{to} specify the behaviours of a passenger respectively in the departure station, going to board a train, and in the arrival station, going to exit the station into the outside world or city.

The initial configuration is:

$$\begin{array}{l}(\nu\, A, B)\, w : g_w(G_w)[\mathsf{c}_w, \mathsf{e}_w \| R \,|\, \text{TRVLRS}(A, B) \\ \qquad |\, A : g_{st}(G_{st})[\mathsf{c}_{st}, \mathsf{e}_{st} \| \,\text{TRAIN}\,] \,|\, B : g_{st}(G_{st})[\mathsf{c}_{st}, \mathsf{e}_{st} \| 0\,] \\ \qquad |\, \text{TRVLRS}(B, A)\,]\end{array}$$

where $\text{TRVLRS}(s, c, d)$ is a parallel composition of processes TRAVELLER.

Our specification satisfies many properties of interest; some of them immediately follow from the definitions. For instance, from the definition of e_{tr} it follows that no traveller can get into the train t when this is outside a station: any action to such purpose from a process in w will be dynamically blocked. Also, by the definition of e_{tr} and of the PSNG process[1] it follows that at most n PSNG processes can be within the train t at the same time.

A d passenger willing to get off the train when this is not in a station, though it maybe statically well typed, is dynamically not allowed to do so. Suppose the bad passenger be represented by the process

$$\text{BADPSNG} \triangleq \mathsf{down}\, t : g_{tr}\, \mathsf{with}\, G_{bad}\, .\, \mathsf{up}\, w : g_w\, \mathsf{with}\, G_w\, .\, \text{BP}$$

By assuming $o; \Sigma \vdash \text{BP} : g_w(G_w)$ one may derive the typing

$$o; \Sigma \vdash \mathsf{up}\, w : g_w\, \mathsf{with}\, G_w\, .\, \text{BP} : g_{tr}(G_{bad})\ \mathsf{with}\ G_{bad} \triangleq \mathsf{mc}(\varnothing, \{g_w\}, \mathsf{shh})$$

Observe that the process type $g_{tr}(G_{bad})$ characterizes a process that might stay within the train and go from it directly into the world. From the above we may infer the typing $o; \Sigma \vdash \text{BADPSNG} : g_{st}(G_{st})$, since for that it is enough that G_{st} allows the process to get into the train, i.e., $g_{tr} \in \mathsf{E}(G_{st})$.

The process BADPSNG is therefore statically allowed to stay within a station, as for example in the well-typed term $A : g_{st}(G_{st})[\mathsf{c}_{st}, \mathsf{e}_{st} \| \,\text{BADPSNG} \,|\, \text{TRAIN}\,]$. Nevertheless, when trying at runtime to get into the train, the process is blocked because $\mathsf{e}_{tr} = \{\langle g_{st}, G_{psng} \rangle^{\mathsf{n}}\}$ (with $G_{psng} \leq G_{tr}$). As a matter of fact, for the action $\mathsf{down}\, t : g_{tr}\, \mathsf{with}\, G_{bad}$ to fire, it is required that $G_{bad} \leq G_{psng}$, which is not the case since $G_{bad} = \mathsf{mc}(\varnothing, \{g_w\}, \mathsf{shh})$ while $G_{psng} = \mathsf{mc}(\varnothing, \{g_{st}\}, \mathsf{shh})$: G_{bad} allows going into the world while G_{psng} does not.

This should have been somehow expected, because in our calculus the dynamic checks performed by an ambient are assigned the task of controlling that mobile processes willing to get in do respect some fixed policies expressed through types and, if this is not the case, of preventing them from getting in. Notice that all the previous properties are guaranteed by only exploiting in the operational semantics information local to the involved processes/ambients.

A similar scenario has already been modelled in [9, 25]. In both cases, the mobility control is implemented by informing the passenger when the train has reached the station at which he wants to get off. More specifically, in [9] a new primitive for n m is exploited. Intuitively, the train ambient takes a suitable name to implicitly inform the passengers when it has arrived at a certain station, while it takes a name unknown to passengers when it is moving (in this way passengers cannot get in or off the train). In [25] a suitable ambient called n is generated by the train when it arrives at a station. This ambient informs the passengers of the arrival at a certain station.

[1] The present specification does not prevent a passenger to add more than one pair to e_{tr}.

3.3 Concluding Remarks

The calculus we presented is a first attempt to model the interplay of static and dynamic type-checking when handling the security requirements of global computing applications. In particular, the packing of a mobility and communication type within a mobile process and its subsequent check at destination may be considered as an abstract modelling of the proof-carrying code approach.

Due to the absence of static ambient types (apart from the atomic type amb), static typing rules may be easily translated into a simple type inference algorithm that, given a term in whose body all the mobcom types are left unspecified, reconstructs the minimal such type allowing the term to be well typed. The algorithm will merely build a type by recording the capabilities occurring in the term. The groups assigned to ambient occurrences, on the other hand, as well as the dynamic components c and e, define the policy and the mobility constraints established by the designer of the application, and cannot be sensibly inferred.

A still unsatisfactory aspect of our model is that the authority (specified by the partial order O) granting dynamic rights to ambients is a too coarse-grain notion: either an ambient is authorized to grant a right with any (even infinite) multiplicity, or the ambient may grant none. It would be useful that this authority could have different degrees, related to maximal multiplicities of granted rights.

Another useful extension would be the introduction of a primitive for group restriction as in [12, 17]. This could provide protection from external untrusted agents, but the interaction with the partial order O representing the administrative hierarchy requires a careful handling. A modification of the calculus in this sense, along with a possible increasing of the expressivity of types, is currently under investigation.

4 Confining Data and Processes in Global Computing Systems

The major contribution of this section is the definition of an approach that permits protecting the secrecy of data residing on hosting nodes and that of data carried by mobile processes by relying on program annotation. Our approach is inspired by Confined-λ [36] and relies on annotating data with sets of node addresses, called *regions*, that specify the network nodes that can interact with them. Also nodes may have annotations that specify which nodes can send data and spawn processes to them. These annotations enable programmers to control the set of nodes that can share specific data, and permit shading them from other nodes. Annotations, instead, enable node administrators to control the set of data and processes each node can host; thus, the node can refuse malicious processes and unwanted data.

The language semantics is then designed to guarantee that computations proceed while respecting the region constraints. For example, a process P can access a datum d only if P's execution does not export d outside the data region, say r, i.e. if P only writes d in network nodes included in r or, similarly, if P only carries d while migrating to nodes included in r. Enforcing similar constraints requires a form of code inspection that is performed, as much as possible, statically thus relieving the runtime semantics of the burden to make expensive checks and, then, improving efficiency. In Section 4.1 we shall introduce more details and a simple motivating example.

Our approach is largely independent of a specific model. Indeed, we shall show how it can be applied to different process calculi for mobile processes. In particular, we shall consider cKLAIM (core KLAIM, a simplified version of KLAIM [19]) in Section 4.2, Dπ (Distributed π-calculus [34], a distributed variant of the π-calculus [42]) in Section 4.3, and Mobile Ambients Calculus [14] in Section 4.4. For each of these calculi, we add regions information to the syntax of terms, define a type system and an operational semantics that take annotations into account, and prove that the semantics guarantees that computations proceed only while respecting confinements. Thus, after successful static type checking, one can guarantee that data are manipulated only by authorised users. Moreover, since in such dynamic environments we cannot assume knowledge of the whole net, we also establish a more general result, namely that absence of violations of data annotations is guaranteed for all successfully type checked sub-nets, regardless of the configuration and of the evolution of the whole net they are in. In Section 4.5 we illustrate our approach

by means of a significative example, where we model the secure implementation of a UNIX-like multiuser system. Comments on the differences between the three typing systems and about future and related work are postponed to Section 4.6.

4.1 Controlling Data Movement via Types

As we have just said, we would like to set up a machinery based on typing that helps in protecting exchanged and local data in global computing applications. To this aim, we suggest annotating data with sets of network addresses, describing the sub-net where data can be used; these sets will be called *regions*. The annotations allow programmers to fix the nodes that can share a given datum, and to avoid that the datum is accessed by untrusted processes (from untrusted nodes). Also network nodes are annotated with regions that specify the nodes that can send data and those that can spawn processes to them. This mechanism allows the administrator of a node to control the data/processes the node can host, and to refuse malicious processes and unwanted data. Thus, nodes are annotated with two regions, say r_d and r_p. We should have $r_p \subseteq r_d$ since accepting processes is, in general, more dangerous than accepting data; however, no restriction on the model is imposed to deal with this issue.

Our typing approach can be implemented by letting *regions* to be either finite subsets of addresses and input parameters or the distinct element $\top$ (used to refer to the whole net). The set of all regions $\mathcal{R}$, ranged over by r, can be partially ordered by the subset inclusion relation $\subseteq$, and has $\top$ as top element. Data annotation is rendered as $[data]_r$; we shall assume that absence of region annotations stands for $\top$.

The language semantics guarantees that computations proceed according to region constraints. This property, that we call *safety*, can be stated as

> A net N is *safe* if, for any datum d occurring in N associated to region r and for all possible evolutions of N, it holds that d will only cross and reside at nodes whose addresses are in r.

To better understand the properties we want to model and the impact of our approach on system security, we present a simple example that, for the time being, is modelled by means of a sort of abstract language (in the following sections, the same example shall be modelled by exploiting each of the calculi we mentioned in the previous section). Operator '.' stands for action prefixing while '$*$' denotes replication of processes. We assume the following process actions:

- $\text{SND}(data, tgt)$ sends the information $data$ by exploiting the communication medium tgt;
- $\text{RCV}(p, tgt)$ receives from the communication medium tgt information that is then bound to parameter p;
- $\text{RES}(n)$ creates a fresh name n, and restricts its visibility to the creating process, shading the name to any other process of the net.

Moreover, we shall use function $a(\_)$ for associating 'access points' to nodes. The exact nature of the access points and of the communication medium tgt, and the way SND and RCV exchange data depends on the chosen communication paradigm. For instance, in case of channel-based synchronous communication, $a(d\ dr_S)$ will return a valid channel for communicating with $d\ dr_S$, tgt will be a suitable communication channel and actions SND/RCV will be executed simultaneously.

For the sake of simplicity, in the following and in the other examples we shall use polyadic communication, although we develop our theoretical framework by considering the monadic and first order variants of the calculi. Here we further rely on remote communications and on a mechanism that permits decomposing received data according to the structure of the parameters specified for receiving them.

Let us now describe the scenario we want to model. Suppose that a client C requires a service to a server S. Once S has verified the credentials of C (e.g. its identity or its credit card information), it sends back a secret password, that C can change. C could then access the service by using the last set password. This

protocol can be modelled by assuming two network addresses, $addr_C$ and $addr_S$, hosting the processes P_C and P_S, respectively, that are defined below.

$$
\begin{array}{lll}
P_C & \triangleq & \mathrm{SND}(addr_C, [credit\,Card\_info]_{\{addr_C;addr_S\}}, addr_S).\\
 & & \mathrm{RCV}(y, addr_S).\langle pwd\,y\,addr_S\rangle\\
P_S & \triangleq & *\,\mathrm{RCV}(x_1, x_2, addr_S).\langle check\,credit\,info\,x_2\rangle.\\
 & & \mathrm{RES}(PWD).\mathrm{SND}([PWD]_{\{x_1;addr_S\}}, addr_S).\\
 & & \langle valid\,pwd\,addr_S\rangle
\end{array}
$$

Notice that, since the information on C's credit card is marked with region $\{addr_C, addr_S\}$, only processes at the locations of C and S will be enabled to capture C's request. Thus, no attacks mounted from other nodes aimed at cancelling the request can take place. Similar considerations do hold for the restricted name PWD that S sends back to C (it represents a secret password shared between processes at C's and S's locations).

To make our theoretical framework properly working, we need to control the processes arriving at C's and S's locations; this is why our typing discipline requires also nodes to be annotated with regions. Server S can then accept only processes coming from trusted nodes, but it should accept data coming from any user; this is necessary to model a setting where S accepts any service request, while it supplies the service only to accredited users. It has to be said that we are implicitly assuming the ability of determining the origin (the source node) of data and processes. By relying on it, we can then check compliance with regions annotations.

In the example above we have exploited remote communication; if only local communication is allowed, we would need to replace $\mathrm{SND}(\ldots, addr_S)$ with something like $\mathrm{EXEC}(\mathrm{SND}(\ldots, addr_S), addr_S)$, supposing that $\mathrm{EXEC}(P, addr)$ spawns process P for execution to the node with address $addr$.

4.2 cKLAIM: Core KLAIM

We start by applying our approach to cKLAIM, a calculus at the core of the language KLAIM (Kernel Language for Agents Interaction and Mobility, [19]). The theory developed here simplifies that of [29] because the calculus only permits monadic communication and uses replication (instead of recursion) to model infinite behaviours.

The syntax of cKLAIM is given in Table 1. There is only one category of names, namely that of *localities*, $\mathcal{L}$, ranged over by l. Identifiers, ranged over by ℓ, can be locality names or variables (ranged over by x), and represent both the communicable data and the target of (possibly remote) actions. T denotes templates for pattern matching and may either be a parameter $!x$, for some variable x, or a locality name. Data are represented as special processes $\langle l\rangle$, thus we may say that each node hosts processes and a (possibly empty) multiset of data. In the following, we assume that in (well formed) processes data are never prefixed by an action or replication. By using LINDA [26] terminology, we shall call *tuple space* (TS, for short) the multiset of data hosted by a node and we let it to represent the repository of the node. Communication can be remote and relies on multiple distributed tuple spaces. cKLAIM nodes are written $l\ ::_{r_p}^{r_d} P$. The two region annotations control the nodes that can send data or processes to l, as established by the node administrator. Process actions are:

- **out** $(\ell')@\ell$: creates a new datum ℓ' (whose region is r) in the TS at ℓ.
- **in** $(T)@\ell$: if $T = !x$, a datum $\langle l\rangle$ is withdrawn from the TS at ℓ and x is replaced by l in the continuation; if $T = \ell'$, then the action will look for (and retrieve) a datum $\langle \ell'\rangle$ at the TS of node ℓ (if any). This second kind of input action is a form of *asynchronous handshake* operator.
- **eval** $(P)@\ell$: spawns process P for execution to the node referred to by ℓ.

$l, h, k, \ldots \in \mathcal{L}$			LOCALITY NAMES
$x, y, z, \ldots$			VARIABLES
ℓ	::=	$l \mid x$	IDENTIFIERS
T	::=	$\ell \mid !x$	TEMPLATES
N	::=	$l\ {}_{r_d}::_{r_p} P \quad \mid \quad N_1 \parallel N_2$	NETS
P	::=		PROCESSES
		$\mathbf{nil}$	(empty)
	$\mid$	$\langle [\![l]\!] \rangle$	(datum)
	$\mid$	$\alpha.P$	(prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	$*P$	(replication)
α	::=		ACTIONS
		$\mathbf{out}\ ([\![\ell']\!])@\ell$	(send)
	$\mid$	$\mathbf{in}\ (T)@\ell$	(receive)
	$\mid$	$\mathbf{eval}\ (P)@\ell$	(execute)
	$\mid$	$\mathbf{newl}\ (l)$	(creation/restriction)

Table 1: CKLAIM Syntax

- $\mathbf{newl}\ (l)$: creates a fresh locality name l that is used as the address of a new node tagged by the region annotations of the creating one and hosting process $\mathbf{nil}$.

Identifiers occurring in process terms can be *bound*. More precisely, prefix $\mathbf{in}\ (!x)@\ell.P$ binds variable x, while $\mathbf{newl}\ (l).P$ binds locality l; in both cases, P is the scope of the binding. An identifier that is not bound is called *free*. We let $\mathrm{fv}(P)$ to denote the set of free variables in P. As usual, *alpha-conversion* allows to freely rename bound identifiers without captures. In the sequel, we shall assume that bound identifiers in processes are all distinct and different from the free ones (this is always possible by using α-conversion). Finally, we shall only consider for execution *closed* nets, i.e. nets where each occurrence of a variable is bound by an $\mathbf{in}$ prefix (similarly to many real compilers, we consider terms with free variables as programming errors). In the rest of the section, we will omit trailing occurrences of the empty process, as usual.

4.2.1 Typing CKLAIM Nets

The language presented in the previous section is a mean to program applications where, during the computation, a datum can only appear in localities contained in its region annotation. The runtime semantics can enforce this requirement by performing appropriate checks. These (runtime) checks are necessary because the pattern matching based communication does not permit making any static assumption on the actual structure of tuples hosted by a tuple space. To make the semantics as efficient as possible, a preliminary typing phase is introduced. Static typing of CKLAIM nets aims at guaranteeing that:

1. a datum $[\![l]\!]_r$ can be seen at (i.e. can cross) ℓ if $\ell \in r$
2. a process retrieving a datum $[\![l]\!]_r$ cannot exhibit l outside r.

The typing phase performs check 1. statically and annotates parameters occurring in templates with regions to enable efficient execution of check 2. at runtime. To better distinguish the annotations put by the programmers/administrators from those put by the type system, we shall write the latter ones as superscripts and the former ones as subscripts. Hence, the syntax of templates becomes

$$T ::= \ell \mid [\![x]\!]^{\Gamma}$$

Typing Nets

(CK-T-NET)
$$\frac{N_1 \succ N_1' \qquad N_2 \succ N_2'}{N_1 \parallel N_2 \succ N_1' \parallel N_2'}$$

(CK-T-NODE)
$$\frac{r_d, r_p \in \{\top\} \cup 2^{\mathcal{L}} \qquad \varnothing \vdash P \succ_l \varnothing \vdash P'}{l\ {}_{r_d}::_{r_p} P \succ l\ {}_{r_d}::_{r_p} P'}$$

Typing Processes

(CK-T-NIL)
$$\frac{}{\Gamma \vdash \mathbf{0} \succ \Gamma \vdash \mathbf{0}}$$

(CK-T-PAR)
$$\frac{\Gamma \vdash P_1 \succ \Gamma' \vdash P_1' \qquad \Gamma' \vdash P_2 \succ \Gamma'' \vdash P_2'}{\Gamma \vdash P_1 | P_2 \succ \Gamma'' \vdash P_1' | P_2'}$$

(CK-T-REPL)
$$\frac{\Gamma \vdash P \succ \Gamma' \vdash P'}{\Gamma \vdash *P \succ \Gamma' \vdash *P'}$$

(CK-T-NEW)
$$\frac{\Gamma \vdash P \succ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{new}\ (l).P \succ \Gamma' \nearrow^l \vdash \mathbf{new}\ (l).P'}$$

(CK-T-DATUM)
$$\frac{l \in r}{\Gamma \vdash \langle [\![l']\!] \rangle \succ_l \Gamma \vdash \langle [\![l']\!] \rangle}$$

(CK-T-MATCH)
$$\frac{\Gamma \vdash P \succ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{in}\ (\ell'')@\ell'.P \succ \Gamma' \vdash \mathbf{in}\ (\ell'')@\ell'.P'}$$

(CK-T-IN)
$$\frac{\Gamma \uplus \{x : \{\ell\}\} \vdash P \succ \Gamma' \uplus \{x : r\} \vdash P'}{\Gamma \vdash \mathbf{in}\ (!x)@\ell'.P \succ \Gamma' \nearrow^x \vdash \mathbf{in}\ ([\![x]\!]^{-\{x\}})@\ell'.P'}$$

(CK-T-OUT)
$$\frac{\{\ell, \ell'\} \subseteq r \qquad \Gamma \vdash P \succ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}\ ([\![\ell'']\!])@\ell'.P \succ \Gamma' + \{x : r\}_{x \in \mathrm{fv}(\ell'')} \vdash \mathbf{out}\ ([\![\ell'']\!])@\ell'.P'}$$

(CK-T-EVAL)
$$\frac{\ell \in reg(P_1) \qquad \Gamma \vdash P_1 \succ_\ell \Gamma' \vdash P_1' \qquad \Gamma' \vdash P_2 \succ \Gamma'' \vdash P_2'}{\Gamma \vdash \mathbf{eval}\ (P_1)@\ell'.P_2 \succ \Gamma'' + \{x : \{\ell'\}\}_{x \in \mathrm{fv}(P_1)} \vdash \mathbf{eval}\ (P_1')@\ell'.P_2'}$$

Table 2: Typing Procedure for CKLAIM

Intuitively, $[\![x]\!]$ states that the datum replacing x will cross at most the localities in r.

The typing procedure for CKLAIM nets is given in Table 2. Net typings are written $N \succ N'$. The typing step includes a *process checking* phase, to verify that nets are written according to the region annotations therein, and a *process inference* phase, to annotate parameters occurring in templates. Intuitively, the inference phase takes a net N (written according to the syntax in Table 1) and returns a net N' obtained from N by annotating all the parameters with a region containing the nodes that the received values will cross. E.g., in process $\mathbf{in}\ (!x)@l.\mathbf{out}\ ([\![x]\!])@l'$ the declaration $!x$ of variable x must be associated to region r. The type checker verifies that each process located at a node l contains only data that can be seen by l (this is done by the judgement $\succ_l$) and verifies that actions **out** and **eval** send data/code to nodes where the data/code can appear without violating the region annotations.

Judgement $\succ$ relies on an auxiliary procedure $\Gamma \vdash P \succ \Gamma' \vdash P'$ where the *process environment* Γ is a finite map from variables to regions such that $\mathrm{fv}(P) \subseteq dom(\Gamma)$. Thus, the procedure $\varnothing \vdash P \succ \varnothing \vdash P'$ is defined only if P is closed; in that case, it determines for each parameter in P a region annotation describing the use of that parameter in the continuation process (i.e. where it will be sent). P' is obtained by decorating P with these annotations. Such regions are determined by the type inference by considering the locality

where the process runs (the ℓ decorating $\succ$) and by examining the localities where the variables can appear upon execution of actions **out** and/or **eval**. Notice, however, that care is needed to avoid that closed nets become open. As an example, consider the nodes (both of them are legal)

$$l :: \mathbf{in}\,(!x)@l'.\mathbf{in}\,(!y)@l''.\mathbf{out}\,(\lfloor x \rfloor_{\{l;y\}})@l \quad (\star)$$
$$l :: \mathbf{in}\,(!y)@l'.\mathbf{out}\,(\lfloor y \rfloor_{\{l;y\}})@l \quad (\star\star)$$

Blindly annotating these nodes would result in

$$l :: \mathbf{in}\,(\lfloor !x \rfloor^{\{l;y\}})@l'.\mathbf{in}\,(\lfloor !y \rfloor^{\{l\}})@l''.\mathbf{out}\,(\lfloor x \rfloor_{\{l;y\}})@l$$
$$l :: \mathbf{in}\,(\lfloor !y \rfloor^{\{l;y\}})@l'.\mathbf{out}\,(\lfloor y \rfloor_{\{l;y\}})@l$$

that are open because of the occurrence of y in the regions of $!x$ and $!y$, respectively. The solution we designed to accept $(\star)$ is to assign $!x$ the region annotation $\top$. This is reasonable since $\mathbf{in}\,(\lfloor !x \rfloor^{\{l;y\}})@l'$ means 'retrieve a datum from l' and share it with *a generic* locality of the net' (indeed y can be dynamically replaced with any locality name). The solution we designed to accept $(\star\star)$ is to remove y from $!y$ region annotation and assume that a locality can always occur in the node having that locality as address.

An anomaly somehow related to $(\star)$ is

$$l :: \mathbf{in}\,(!x)@l'.\mathbf{newloc}\,(l'').\mathbf{out}\,(\lfloor x \rfloor_{\{l;l''\}})@l \quad (\dagger)$$

that would result in the annotated process

$$l :: \mathbf{in}\,(\lfloor !x \rfloor^{\{l;l''\}})@l'.\mathbf{newloc}\,(l'').\mathbf{out}\,(\lfloor x \rfloor_{\{l;l''\}})@l$$

Here the problem is that the l'' occurring in the annotation associated to $!x$ by the inference system escapes from the binder **newloc** that declares l''. Thus, these two occurrences of l'' are not the same! For the sake of simplicity, we overcome this problem like before, i.e. by assigning $\top$ to the region annotation of $!x$.

To rule out anomalies like $(\star)$ and $(\dagger)$, in Table 2 we use function $\Gamma\nearrow^{\ell}$, that is inductively defined as

$$\begin{array}{rcll} \varnothing\nearrow^{\ell} & = & \varnothing & \\ (\Gamma \uplus \{x : r\})\nearrow^{\ell} & = & \Gamma\nearrow^{\ell} \uplus \{x : \top\} & \text{if } \ell \in r \\ & & \Gamma\nearrow^{\ell} \uplus \{x : r\} & \text{otherwise} \end{array}$$

where $\uplus$ denotes union between environments with disjoint domains.

Function $+$ extends the information of an environment through another environment and is undefined if the domain of the second environment is not included in that of the first one; formally

$$\begin{array}{rcll} \Gamma + \varnothing & = & \Gamma & \\ \Gamma + \{x : r\} & = & \Gamma' \uplus \{x : r \cup r'\} & \text{if } \Gamma = \Gamma' \uplus \{x : r'\} \\ \Gamma + (\{x : r\} \uplus \Gamma') & = & (\Gamma + \{x : r\}) + \Gamma' & \end{array}$$

Finally, we write $\{\ldots\}_{i\in I}$ to mean $\bigcup_{i\in I}\{\ldots\}$.

Before concluding this section, we briefly comment on some typing rules. Notice that the typing of N also verifies that N is closed. Moreover, it can be easily seen that typing $P_1|P_2$ and $P_2|P_1$ yields the same typing; this relies on commutativity of sets union, since Γ grows up by union of regions. In rule (CK-T-NEW), the resulting environment is $\Gamma'\nearrow^{l}$ to rule out anomalies like $(\dagger)$. In rule (CK-T-IN), the procedure should type P in the environment Γ extended by associating x to region $\{\ell\}$. At the end of this typing phase, the region annotation r calculated for x is associated to the parameter $!x$. Notice that x can occur in x's region r, generating anomalies like $(\star\star)$; to avoid this, the annotation for x must be $r - \{x\}$. Moreover, it is possible that x occurs in region annotations within Γ' because of anomalies like $(\star)$; thus, the environment resulting from this phase must be $\Gamma'\nearrow^{x}$. In rule (CK-T-OUT), the type checker verifies

that ℓ'' can stay both in the hosting locality ℓ and in the target locality ℓ'. The continuation process P is typed in the environment Γ, thus obtaining the annotated process P' and the environment Γ'. Hence, the result of the typing will be $\mathbf{out}(\ldots)[\ell'']_{\ell'}@\ell'.P'$ together with Γ' extended with the information that the variables occurring in ℓ'' (i.e. x if $\ell'' = x$) could be seen at r. Similar observations also hold for rule (CK-T-EVAL) too; in particular, the check that the process can cross the locality where it is hosted is performed whenever the process is going to migrate. To this aim, we exploit the auxiliary function $reg(\_)$ that returns the intersection of the data regions occurring in its argument. Its formal definition is

$$\begin{array}{rcl@{\qquad}rcl} reg(\mathbf{halt}) & = & \top & reg(\mathbf{new}\ (l).P) & = & reg(P) - \{l\} \\ reg(\mathbf{out}\ (\ldots)[\ell']_{\ell}@\ell.P) & = & r \cap reg(P) & reg(\mathbf{in}\ (T)@\ell.P) & = & reg(*P) \sqcap reg(P) \\ reg(P_1|P_2) & = & reg(\mathbf{eval}\ (P_1)@\ell.P_2) \sqcap reg(P_1) \cap reg(P_2) & & & \end{array}$$

We deem $\mathbf{halt}$ those nets that successfully passed a typing phase.

Definition 4.1 (Well-typed CKLAIM Net). *A net N is* well-typed *if there exists a net N' such that $\vdash N \rightarrow N'$ holds. The net N' is such that $N' \succ N$.*

4.2.2 CKLAIM Type Checking

CKLAIM nets are executed according to the reduction relation $\succ\!\!\rightarrow$ defined in Table 3. $\succ\!\!\rightarrow$ relates configurations of the form $L \triangleright N$, where L is such that $loc(N) \subseteq L \subset_{fin} \mathcal{L}$ and function $loc(N)$ returns the set of localities occurring in N. In a configuration $L \triangleright N$, L is needed to ensure global freshness of new addresses. For the sake of readability, when a reduction does not generate any fresh addresses, we write $N \succ\!\!\rightarrow N'$ instead of $L \triangleright N \succ\!\!\rightarrow L \triangleright N'$. We denote with $L \uplus L'$ the disjoint union of sets L and L', and with $\succ\!\!\rightarrow^?$ the reflexive and transitive closure of $\succ\!\!\rightarrow$.

The semantics exploits *substitutions*, replacing variables with locality names; the substitution mapping x to l will be written as $\{^l/_x\}$. The application of $\{^l/_x\}$ to any syntactic term (variable/region/process/type environment) t, denoted by $t\{^l/_x\}$, replaces each free occurrence of x in t with l, with renaming of bound variables possibly involved to avoid captures. We remark that the application of a substitution to a process P also acts on the region annotations in P.

The reduction relation relies on a *structural congruence* relation, $\equiv$, equating α-convertible processes, stating that "$||$" is commutative and associative, and that $\mathbf{halt}$ acts as the identity for "$|$".

We now comment on the semantics rules. Rules (CK-OUT) and (CK-EVAL) say that a datum/process can be put at the target of the $\mathbf{out}$/$\mathbf{eval}$ only if such a node accepts the datum/process (i.e. $l \in r'_d$ and $l \in r'_p$). This is necessary to prevent an untrusted node l to send data/code to l'. Notice that no static check could enforce this property without loss of expressivity: e.g., in $\mathbf{in}\ (!x)@l.\mathbf{eval}\ (\ldots)@x$, it is statically impossible to know which locality will replace x without limiting the possible exchanges at l. Thus, it cannot be determined if the locality executing the $\mathbf{eval}$ is trusted by the target locality or not. Rule (CK-IN) says that a process can retrieve a datum only if the continuation process respects the datum annotation (i.e. $r' \subseteq r$). If a datum is present in the target of the action for which this check succeeds, then the datum is retrieved and replaces the input variable in the continuation; otherwise, the process is suspended until such a datum is available (if ever). Rule (CK-MATCH) verifies if a datum l'' is present in l'. If this is the case, the datum is removed and the continuation proceeds; otherwise, the process is suspended. Notice that, in order to complete this task, the node executing the action must be authorised by the region r. In rule (CK-NEW) the set L of localities already in use is exploited to choose a fresh address l' for naming the new node. Moreover, we assume that a node l trusts every node l' it creates. This is reasonable since, once created, l' is not known to any other node in the net; thus, l can use it as a *private* resource and can decide the nodes of the net that can know it (by also exploiting region annotations). For the sake of simplicity, l' is assigned the trust regions of l. However, it would be easy to extend the language for allowing the programmer to explicitly specify the

(cK-OUT)
$$\frac{l \in r'_d}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{out}\ (\![l'']\!).P \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ P' \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ P' \mid \langle (\![l'']\!) \rangle}$$

(cK-EVAL)
$$\frac{l \in r'_p}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{eval}\ (Q)@l'.P \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ P' \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ P' \mid Q}$$

(cK-IN)
$$\frac{r' \subseteq r}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}\ (\![x]\!)@l'.P \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ \langle (\![l'']\!) \rangle \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ P\{l''/x\} \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ \mathbf{0}}$$

(cK-MATCH)
$$\frac{l \in r}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}\ (l'')@l'.P \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ \langle (\![l'']\!) \rangle \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ \mathbf{0}}$$

(cK-NEW)
$$\frac{}{L \triangleright l\ {}_{r_d}{::}_{r_p}\ \mathbf{new}\ (l').P \succ\!\!\rightarrow L \uplus \{l'\} \triangleright l\ {}_{r_d \cup \{l^0\}}{::}_{r_p \cup \{l^0\}} P \parallel l'\ {}_{r_d \cup \{l^0\}}{::}_{r_p \cup \{l^0\}}\ \mathbf{0}}$$

(cK-CALL)
$$\frac{}{l\ {}_{r_d}{::}_{r_p}\ *P \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ *P \mid P}$$

(cK-SPLIT)
$$\frac{L \triangleright l\ {}_{r_d}{::}_{r_p}\ P_1 \parallel l\ {}_{r_d}{::}_{r_p}\ P_2 \parallel N \succ\!\!\rightarrow L' \triangleright l\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ P'_1 \parallel l\ {}_{r_d}{::}_{r_p}\ P'_2 \parallel N'}{L \triangleright l\ {}_{r_d}{::}_{r_p}\ P_1 \mid P_2 \parallel N \succ\!\!\rightarrow L' \triangleright l\ {}_{r'^{\bullet}_d}{::}_{r'^{\bullet}_p}\ P'_1 \mid P'_2 \parallel N'}$$

(cK-PAR)
$$\frac{L \triangleright N_1 \succ\!\!\rightarrow L' \triangleright N'_1}{L \triangleright N_1 \parallel N_2 \succ\!\!\rightarrow L' \triangleright N'_1 \parallel N_2}$$

(cK-STRUCT)
$$\frac{N_1 \equiv N'_1 \qquad L \triangleright N'_1 \succ\!\!\rightarrow L' \triangleright N'_2 \qquad N'_2 \equiv N_2}{L \triangleright N_1 \succ\!\!\rightarrow L' \triangleright N_2}$$

Table 3: cKLAIM Operational Semantics

trust regions of a newly created node. Rule (cK-CALL) unfolds a replicated process and corresponds to a procedure call. Rule (cK-SPLIT) permits splitting the parallel processes running at a node thus enabling the application of the main reduction rules that, in fact, can be used when there is only one thread running at l. Technically, a parallel between processes is transformed into a parallel between nodes. Rules (cK-PAR) and (cK-STRUCT) are standard: the former says that, if part of a composed net evolves, the whole net evolves accordingly and the latter says that structural congruent nets have the same reductions.

We now give two simple properties of the operational semantics. The first one describes the relationship between the set of localities L in a configuration $L \triangleright N$ and the localities occurring in the net obtained after a reduction step. The second one describes the way parallel components located at node l could have been arrived there: they could have been either allocated at l in the initial setting or placed at l by authorised nodes as the result of subsequent computations.

Proposition 4.2 *If $L \triangleright N \succ\!\!\rightarrow L' \triangleright N'$ then $loc(N) \subseteq L$ and $loc(N') \subseteq L'$.*

P p **4.3** *L* $L \triangleright N \succ\!\!\longrightarrow L' \triangleright N'$, $l \notin L' - L$ *d* $l\ {}_{r_d}\!::_{r_p} P$ *b a d* *f* N'. *T f y p h p n* P' *in* P *t l h* (i) *h* P' *w l* *t* l *in h h og b* N, *o* $(i\ )P'$ *is a th w* *t* l *b a d* *in* r_d, *o* $(i\ \ )P'$ *is a p o p v* *o* l *b a d* *in* r_p.

To conclude this section, we implement in cKLAIM the example presented in Section 4.1. In this setting, the addresses are l_C and l_S with region annotations such that $l_S \in r_d^C$ and $l_C \in r_d^S$ (usually, $r_d^S = \top$), while $r_p^C = r_p^S = \varnothing$. Processes P_C and P_S become

$$
\begin{aligned}
P_C \quad & \mathbf{out}\ (l_C, [c\_info]_{\{l_C;l_S\}})@l_S.\mathbf{in}\ (!y)@l_C.\\
& < \textit{ch ify p v d y d e th e e} >\\
P_S \quad & *\mathbf{in}\ (!x_1, !x_2)@l_S. < \textit{ch k c d it a d in fo } x_2 > .\\
& \mathbf{w}\ (p).\mathbf{out}\ ([p]_{\{x_1;l_S\}})@x_1.\\
& < \textit{l le p v ch if a t o d p oi e th e e} >
\end{aligned}
$$

By reasonably assuming that the password modification is carried on by only involving l_C and l_S, the inference system annotates P_C as follows:

$$
P'_C \quad \mathbf{out}\ (l_C, [c\_info]_{\{l_C;l_S\}})@l_S.\mathbf{in}\ ([!y]^{\{l_C;l_S\}})@l_C.\cdots
$$

Similarly, if we assume that credit card checking is performed locally by the server and never used anymore, P_S is annotated as:

$$
P'_S \quad *\mathbf{in}\ (!x_1, [!x_2]^{\{l_S\}})@l_S.\cdots.\mathbf{w}\ (p).\mathbf{out}\ ([p]_{\{x_1;l_S\}})@x_1.\cdots
$$

Now, the dynamic checks of rule (cK-In) are respected; thus, the resulting net can evolve as expected:

$$
\begin{aligned}
& l_C\ {}_{r_d^C}\!::_{r_p^C} P'_C \parallel l_S\ {}_{r_d^S}\!::_{r_p^S} P'_S\\
\succ\!\!\longrightarrow^{?}\ & l_C\ {}_{r_d^C}\!::_{r_p^C} \mathbf{in}\ ([!y]^{\{l_C;l_S\}})@l_C.\cdots \parallel\\
& l_S\ {}_{r_d^S}\!::_{r_p^S} P'_S \mid < \textit{ch k c\_in fo} > .\mathbf{w}\ (p).\mathbf{out}\ ([p]_{\{l_C;l_S\}})@l_C.\cdots\\
\succ\!\!\longrightarrow^{?}\ & l_C\ {}_{r_d^C}\!::_{r_p^C} < \textit{ch ify p v d p d e th e e} > \parallel\\
& l_S\ {}_{r_d^S}\!::_{r_p^S} P'_S \mid < \textit{l le p v ch if a t o d p oi e th e e} >
\end{aligned}
$$

Notice that, in the reductions above, we omitted the sets L of localities in use: they can be easily inferred. Moreover, as usual, we used $\succ\!\!\longrightarrow^{?}$ to denote the reflexive and transitive closure of $\succ\!\!\longrightarrow$.

4.2.3 T p l

Our main results state that well-typedness is preserved along reductions and that well-typed nets do respect region annotations. The former result is called *b r l*; the latter result is called *p f* and states that well-typedness guarantees that there are no immediate violations of data regions. Together, these results imply the *d* of our theory, i.e. no violation of data regions will ever occur during the evolution of well-typed nets.

We start with the classical 'soundness' result for any type system, namely subject reduction.

T m 4.4 (S R *f* N *is p v d* $L \triangleright N \succ\!\!\longrightarrow L' \triangleright N'$ *h* N' *is p v*

We now turn to type safety. As we have already said, it states that well-typedness guarantees absence of immediate violations of data regions. However, the wanted safety property requires that data regions are respected along all possible computations. To properly formalise this property we need to define a finer semantics. Indeed, deeming a net to be safe when "for any node $l\ {}_{r_d}\!::_{r_p} P$ it holds that l occurs in the region of each datum in P" would not be satisfactory because the regions annotating data disappear upon data

withdrawal. Thus, it would become impossible to formalise the requirement that the region specification associated to a datum when it is produced is respected during all the datum life-time (i.e. also after its retrieval). For example, consider the net $N = l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}\ (\lambda x)^{0}@l'.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \langle [l'']_{r}\rangle$. Upon execution of action $\mathbf{in}$, the net becomes $N' = l\ {}_{r_d}{::}_{r_p}\ P' \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \mathbf{0}$, where $P' = P\{l''/x\}$. Now, all the occurrences of l'' in P' are not annotated anymore with region r. Hence, in N' we have no mean to formalise the statement that l can use l'' by respecting the original annotation r.

In other terms, with the calculus introduced in Table 1, we cannot express absence of data regions violations syntactically, because in general we lack information about the region that originally annotated data carried by processes. To overcome this problem, we design a *tagged* variant of the calculus, where each occurrence of a locality in a process is tagged with a region determining its visibility. To this aim, we slightly adapt the syntax of CKLAIM, by letting identifiers be

$$\ell ::= [l]_{r} \mid x$$

We can now formalise when a net is safe. To this aim, we extend function *reg* defined in Section 4.2.1 by taking into account also the locality tags when calculating the region intersection. For example, $reg(\mathbf{out}\ (t)_{r_1}@[l]_{r_2}.P) = r_1 \cap r_2 \cap r_3 \cap reg(P)$. Moreover, we let $reg(\langle [l]_{r}\rangle) = r$.

Definition 4.5 (Safe net). *A net N is* safe *if for every subnet $l\ {}_{r_d}{::}_{r_p}\ P$ in N, it holds that $l \in reg(P)$.*

The tagged semantics generalises that in Table 3. Indeed, processes like $\mathbf{out}\ (t)_{r_1}@[l]_{r_2}$ or $\mathbf{in}\ (x)_{r_1}@[l']_{r_2}$ can evolve. These terms may arise upon application of substitutions that now map variables into localities tagged with regions. We let the application of the substitution to a region to replace variables only with localities (hence omitting their tags) thus ensuring that regions are still sets of identifies. The reduction relation, however, ignores the tags and considers tagged names as plain ones. This should have been somehow expected because, as we said before, the only role of tags is to enable formalising and checking that a net is safe. Thus, rules (CK-OUT) and (CK-IN) now become

$$\frac{l \in r'_d}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{out}\ (t)_{r''}@[l']_{r'}.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P' \ \succ\!\!\longrightarrow\ l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P' \mid \langle [t]_{r''}\rangle}$$

$$\frac{r_1 \subseteq r}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}\ (x)_{r''}@[l']_{r'}.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \langle [l'']_{r}\rangle \ \succ\!\!\longrightarrow\ l\ {}_{r_d}{::}_{r_p}\ P\{[l'']_{r}/x\} \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \mathbf{0}}$$

To avoid confusion, we use the arrow $\succ\!\!\longrightarrow$ to relate tagged terms. The other rules extend those in Table 3 in the expected way.

The typing procedure for tagged terms is denoted by $\succcurlyeq$ and its most significant rules are given in Table 4 (the other ones are smooth adaptations of those in Table 2). We use functions $pl(\ell)$ and $reg(\ell)$ to denote, respectively, the plain identifier and the region of the tagged identifier ℓ. The intuition underlying $\succcurlyeq$ is that, whenever an identifier occurs at a locality, the locality must be included in the region tagging the identifier.

Given a plain net N, we use $tag(N)$ to denote the set containing all the well-typed (w.r.t. $\succcurlyeq$) tagged nets obtained by tagging localities in N. Given a tagged net N, we denote with $untag(N)$ the plain net obtained from N by removing all the locality tags. Notice that $tag(N)$ is not empty because it contains at least the net obtained by tagging each locality in N with $\top$. We call the latter net the *totally tagged* net of N.

Predictably, the tagged language and the original one are strongly related. Moreover, the typing of tagged terms is preserved along (tagged) reductions. The following results formalises these properties.

Proposition 4.6

1. *If $N \succcurlyeq M$ then $untag(N) \succ untag(M)$.*

$$\frac{\ell \in reg(\ell') \cap reg(\ell'') \qquad \Gamma \vdash P \succ\!\!\succ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{in}\ (\ell'')@\ell'.P \succ\!\!\succ \Gamma' \vdash \mathbf{in}\ (\ell'')@\ell'.P'}$$

$$\frac{\ell \in reg(\ell') \qquad \Gamma \uplus \{x : \{\ell\}\} \vdash P \succ\!\!\succ \Gamma' \uplus \{x : r\} \vdash P'}{\Gamma \vdash \mathbf{in}\ (!x)@\ell'.P \succ\!\!\succ \Gamma' \nearrow^{x} \vdash \mathbf{in}\ ([!x])^{-\{x\}}@\ell'.P'}$$

$$\frac{\ell \in reg(\ell') \qquad \{\ell, \ell'\} \subseteq r \qquad \Gamma \vdash P \succ\!\!\succ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}\ ([x])@\ell'.P \succ\!\!\succ \Gamma' + \{x : r\} \vdash \mathbf{out}\ ([x])@\ell'.P'}$$

$$\frac{\ell \in reg(\ell') \qquad \{\ell, \ell'\} \subseteq r_2 \subseteq r_1 \qquad \Gamma \vdash P \succ\!\!\succ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}\ ([l]_1]_2)@\ell'.P \succ\!\!\succ \Gamma' \vdash \mathbf{out}\ ([l]_1]_2)@\ell'.P'}$$

$$\frac{\ell \in reg(\ell') \cap reg(P_1) \qquad \Gamma \vdash P_1 \succ\!\!\succ_0 \Gamma' \vdash P_1' \qquad \Gamma' \vdash P_2 \succ\!\!\succ \Gamma'' \vdash P_2'}{\Gamma \vdash \mathbf{eval}\ (P_1)@\ell'.P_2 \succ\!\!\succ \Gamma'' + \{x : \{\ell\}\}_{x \in \mathrm{fv}(P_1)} \vdash \mathbf{eval}\ (P_1')@\ell'.P_2'}$$

Table 4: Tagged Typing Rules

2 *If* $N \succ M$, *then for all* $M' \in tag(M)$ *there exists* $N' \in tag(N)$ *such that* $N' \succ\!\!\succ M'$.

3 *If* $L \triangleright N \succ\!\!\!\rightarrow\!\!\!\rightarrow L' \triangleright N'$ *then* $L \triangleright tag(N) \succ\!\!\!\rightarrow L' \triangleright tag(N')$.

Corollary 4.7 (Tagging preserves well-typedness) *If* N *is a well-typed net and* tag *a tagging and* $L \triangleright N \succ\!\!\!\rightarrow\!\!\!\rightarrow L' \triangleright N'$ *then* N' *is a well-typed net and* tag *a tagging*.

We are now ready to state the type safety theorem.

Theorem 4.8 (Type safety) *If* N *is a well-typed net and* tag *a tagging then* N *is safe*.

Corollary 4.9 (Type safety) *Let* N *be a net and* N' *be its* tag*-tagging. If* $L \triangleright N' \succ\!\!\!\rightarrow\!\!\!\rightarrow^{?} L' \triangleright N''$ *then* N'' *is safe*.

The results given above can be generalised by requiring only a subnet of the whole net to be well-typed. By using the convention that absence of a region annotation means $\top$, a not well-typed net can be executed according to the (tagged versions of) rules in Table 3 by safely considering all its variable annotations as $\top$. We call *region* r *of* N the net formed by all the nodes $l\ {}_{r_d}::_{r_p} P$ in N such that $\{l\} \cup r_d \cup r_p \subseteq r$. Notice that such a net is not necessarily defined for all r; of course it is always defined for $r = \top$ and coincides with N (in this case Theorem 4.10 coincides with Corollary 4.9).

Theorem 4.10 (Local Type safety) *Let* N *be a net and* N' *be its* tag*-tagging. If the region* r *of* N' *is closed and well-typed, and if* $L \triangleright N' \succ\!\!\!\rightarrow\!\!\!\rightarrow^{?} L' \triangleright N''$, *then the region* r' *of* N'' *is closed and well-typed, where* $r' = r \cup (L' - L)$.

To conclude, we want to remark that the language can be easily extended to enable explicit specification of the regions of the new nodes. In this case, existence of the r'-subnet could be ensured by adding a premise to rule (CK-NEW) requiring that the regions of the new nodes are included in those of the creating node.

$l, h, k, \ldots \in \mathcal{L}$			LOCALITY NAMES
$a, b, c, \ldots \in \mathcal{C}$			CHANNEL NAMES
$x, y, z, \ldots$			VARIABLES
e	::=	$l \mid a$	NAMES
u	::=	$e \mid x$	IDENTIFIERS
X	::=	$x \mid z@y$	INPUT PARAMETERS
W	::=	$u \mid v@u$	MESSAGES
N	::=	$l^{r_p}_{r_d}[\![P]\!] \mid N_1 \parallel N_2$	NETS
P	::=		PROCESSES
		$\mathbf{0}$	(empty)
	$\mid$	$\alpha.P$	(prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	$(\nu e)P$	(restriction)
	$\mid$	$*P$	(replication)
α	::=		ACTIONS
		$u!\langle [\![W]\!]_r \rangle$	(send)
	$\mid$	$u?(X)$	(receive)
	$\mid$	$\mathbf{go}\ u$	(migrate)

Table 5: Dπ Syntax

4.3 Dπ: Distributed π-calculus

We now apply our approach to Dπ [34], a variant of the π-calculus [42] with process distribution and mobility. The syntax of Dπ is given in Table 5. There are two categories of names: *localities*, ranged over by l, and *channels*, ranged over by a. The symbol e is used for channel or locality names, while u, v, called *identifiers*, denote names and variables (ranged over by x). The exchanged messages, ranged over by W, can be both identifiers and compound identifiers of the form $v@u$ (where u is expected to be a locality name or variable, while v is expected to be a channel name or variable). Similarly, input parameters, generically referred to as X, can either be a simple variable x or a compound variable $z@y$ (y is a locality variable and z is a channel variable). Dπ nodes (*sites*, in the original terminology) will be written as $l^{r_p}_{r_d}[\![P]\!]$. Communication is local, synchronous and channel based. Process actions are:

- $u!\langle [\![W]\!]_r \rangle$: makes available message W (with associated region r) along the channel u of the locality where the action is fired.
- $u?(X)$: retrieves a message W from the channel u of the locality where the action is fired and replaces the parameter X with the message in the continuation process. If X is a variable x, the message retrieved must be a name e. Otherwise, if X is $z@y$, then the message must be of the form $a@l$, and z will be replaced by a and y will be replaced by l.
- $\mathbf{go}\ u$: spawns the continuation process for execution at the node referred to by u.

Identifiers occurring in process terms can be *bound*; more precisely, prefix $u?(X).P$ binds the variables in X (i.e. it binds x if $X = x$ and binds both y and z if $X = z@y$), while $(\nu e)P$ binds name e; in both cases, P is the scope of the binding. The set of free variables $\mathrm{fv}(\_)$, α-conversion and closed nets are defined accordingly.

To conclude the presentation, we want to argue for the need to associate two regions to each Dπ node. Indeed, differently from CKLAIM, no remote operation is allowed (a part, of course, process spawning),

hence the data region could seem useless. However, using only the process region would be too restrictive: in fact, if a node l does not know or trust another node k, then k has no mean to come into contact with l. Our solution permits to distinguish generic processes from processes that are not very risky because, for example, they only perform an output and then terminate. These last processes are of the form $u!\langle W\rangle.\mathbf{0}$ and we deem them *p* *p* *s*. However, processes of different form could be accepted as well: e.g. process $u!\langle W\rangle.v!\langle W'\rangle.\mathbf{0}$ is as risky as $u!\langle W\rangle.\mathbf{0}$. Since we do not want to take a definite standing on the set of output processes, we use a predicate *o* *tp* $t(P)$, that holds true if and only if P is an output process, but leave aside its exact definition. Thus, output processes coming from k are accepted by node $l^{r_p}_{r_d}[\![\,:\,]\!]$ only if $k \in r_d$; all the other processes are accepted if $k \in r_p$ (see rule (D-T-GO) in Table 6).

4.3.1 Typing Dπ Nets

The typing system for CKLAIM of Table 2 could be straightforwardly adapted to deal with Dπ nets; see Remark 1 in Section 4.3.2. However, in a channel-based setting region compatibility checks can be statically performed (on the contrary, they are dynamically performed in CKLAIM – see rule (CK-IN)) because it is natural to associate each channel with a region annotation describing the region of the data exchangeable along it. Thus, if a channel a can carry data visible within r, then messages with region $r_1 \supseteq r$ can be sent along a and input parameters with region $r_2 \subseteq r$ can be used to retrieve data from a. By transitivity, we get $r_2 \subseteq r_1$ thus ensuring that the use of the data respects the specifications of the data region. Hence, in this setting, parameters do not need to be annotated because the correct use of the data can be statically enforced by the typing system.

To properly deal with name passing, we take advantage of some of the theory from [34]. The resulting type system is very different from that in Table 2 but shows how our approach can be adapted to different languages. We assume the following types:

TYPES:	τ	::=	$\phi \mid \gamma \mid \gamma@\phi$
LOCALITY TYPES:	ϕ	::=	$r \triangleright r'[\mu, \gamma]^{p}_{d}$
CHANNEL TYPES:	γ	::=	$r(\tau)$

Intuitively, if v has type $r(\tau)$ then it is a channel that can be seen by nodes in r and can carry messages of type τ. Similarly, if v has type $r \triangleright r'[\mu, \gamma]^{p}_{d}$ then it is a locality whose name can be one of the names in r' (this is useful only when v is a variable; if v is a name, then $r' = \{v\}$, see requirement (‡) below), that can be seen by nodes in r, accepts data/code from nodes in r_d/r_p resp., and hosts channels *e*, in an orderly way of types *e*. As usual, *e* denotes a (possibly empty) set of entities _. Finally, $\gamma@\phi$, with $\phi = \triangleright r[\![\,r' r_d r_p u, \gamma$, can be assigned to a message $u@v$ where u is a channel of type γ and v is a locality of type $r \triangleright r'[\mu, \gamma, u : \gamma]^{p}_{d}$.

For a type τ, we let $reg(\tau)$ to denote the region that can see values of type τ, i.e. $reg(r(\tau) = reg(r \triangleright r'[\mu, \gamma]^{p}_{d}) \quad r$. Similarly, for a locality type $\phi = r \triangleright r'[\mu, \gamma]^{p}_{d}$, we let *l* (ϕ), $dreg(\phi)$ and *p* $eg(\phi)$ to denote, resp., regions r', r_d and r_p.

The typing system for Dπ nets is given in Table 6. The main judgement is $\Gamma \vdash N$, stating that N is well-typed in the environment Γ. A *p* *e* *v* *m* is a finite partial function mapping locality names and variables to locality types. Therefore, since locality types contain information about the allocated channels, it is also possible to extract from a typing environment the channel types associated to channel names and variables. In particular, if $\Gamma(u) = r \triangleright r'[v : \gamma, \widetilde{v' : \gamma'}]^{p}_{d}$, we shall write $\Gamma(u, v) = \gamma$. We shall only consider typing environments satisfying the following constraint:

> Let $\Gamma(v) = r \triangleright r'[\mu, \gamma]^{p}_{d}$. If $v \in \mathcal{L}$ then $r' = \{v\}$; otherwise, for each $l \in r'$, it must be that $r \subseteq reg(\Gamma(l))$ and $r_d \subseteq dreg(\Gamma(l))$ and $r_p \subseteq p\ eg(\Gamma(l))$. (‡)

This condition states that the component r' is really useful only when v is a variable; in this case, it collects the possible names v can assume at runtime. Moreover, it states that regions $r/r_d/r_p$ must respect the corresponding specifications contained in Γ for all the values v can assume.

Typing Networks

$$\text{(D-T-NET)}\ \frac{\Gamma \vdash N_1 \qquad \Gamma \vdash N_2}{\Gamma \vdash N_1 \parallel N_2} \qquad \text{(D-T-NODE)}\ \frac{\Gamma(l) = r \triangleright r' [\alpha \gamma]^{\rho}_{\delta} \qquad \Gamma \vdash_l P}{\Gamma \vdash l^{r}_{r'}[\![P]\!]}$$

Typing Processes

$$\text{(D-T-NIL)}\ \frac{}{\Gamma \vdash_u \mathbf{0}} \qquad \text{(D-T-REPL)}\ \frac{\Gamma \vdash_u P}{\Gamma \vdash_u *P} \qquad \text{(D-T-PAR)}\ \frac{\Gamma \vdash_u P_1 \qquad \Gamma \vdash_u P_2}{\Gamma \vdash_u P_1 | P_2}$$

$$\text{(D-T-CRES)}\ \frac{\Gamma, {}_u a : \gamma \vdash_u P}{\Gamma \vdash_u (\nu a) P} \qquad \text{(D-T-LRES)}\ \frac{\Gamma, {}_u l : \top \triangleright \{l\} [\varnothing]^{preg(\Gamma(u)) \cup \{l\}}_{dreg(\Gamma(u)) \cup \{l\}} \vdash_u P}{\Gamma \vdash_u (\nu l) P}$$

$$\text{(D-T-IN)}\ \frac{\Gamma \vdash_u v : r(\tau) \qquad \mathbf{if}\ (\Gamma(u) \subseteq r \qquad \Gamma, {}_u X : \tau \vdash_u P}{\Gamma \vdash_u v?(X).P}$$

$$\text{(D-T-OUT)}\ \frac{\Gamma \vdash_u v : r'(\tau) \qquad \mathbf{if}\ (\Gamma(u) \subseteq r' \qquad \Gamma \vdash_u W : \tau' \qquad \tau \sqsubseteq \tau' \qquad u \in r \qquad \bigcup_{w \in r} \mathbf{if}\ (\Gamma(w) \subseteq reg(\tau) \qquad \Gamma \vdash_u P}{\Gamma \vdash_u v!\langle [W] \rangle.P}$$

$$\text{(D-T-GO)}\ \frac{\mathbf{if}\ (\Gamma(u) \subseteq reg^{u}_{\Gamma}(\mathbf{go}\ v.P) \quad \text{if } \alpha\ tp\ t(P) \text{ then } \mathbf{if}\ (\Gamma(u) \subseteq dreg(\Gamma(v)) \text{ else } \mathbf{if}\ (\Gamma(u) \subseteq preg(\Gamma(v)) \qquad \Gamma \vdash_v P}{\Gamma \vdash_u \mathbf{go}\ v.P}$$

Typing Messages

$$\text{(D-T-CHAN)}\ \frac{\Gamma(u(v)) = \gamma}{\Gamma \vdash_u v : \gamma} \qquad \text{(D-T-LOC)}\ \frac{\Gamma(v) = \phi}{\Gamma \vdash_u v : \phi} \qquad \text{(D-T-COMPOUND)}\ \frac{\Gamma(w) = r \triangleright r' [\widetilde{v' : \gamma'}, v : \gamma]^{\rho}_{\delta}}{\Gamma \vdash_u v@w : \gamma@r \triangleright r' [\widetilde{v' : \gamma'}]^{\rho}_{\delta}}$$

Table 6: Type Checking for Dπ

We assume that $\Gamma \vdash N$ holds true only if Γ satisfies ($\ddagger$), Γ does not contain variables and $\mathrm{fv}(N) = \varnothing$. The main judgement relies on two auxiliary judgements for typing processes and messages. Judgement $\Gamma \vdash_u P$ states that P can be properly executed at u while respecting Γ; we always assume that $\mathrm{fv}(P) \subseteq dom(\Gamma)$. Judgement $\Gamma \vdash_u W : \tau$ states that message W can be assigned type τ at u under the assumptions Γ. Some aspects, like the extension of an environment with a new item (written $\Gamma, {}_u W : \tau$) and the subtyping relation (written $\tau \sqsubseteq \tau'$), have been straightforwardly adapted from [34] and are given in Table 7. We omit comments on these features and refer the interested reader to [34].

We now briefly comment on some of the typing rules. In rule (D-T-LRES), we assume that the created node is assigned the regions of the creating one (this is similar to CKLAIM – see rule (CK-NEW)). In rule (D-T-IN), it is checked that u can access channel v (the fact that v is a channel is ensured by the fact that Γ indirectly assigns v a channel type through the type of the locality where v is placed) and that the continuation properly uses the received message (i.e. P is typeable in an environment obtained by extending Γ with the information that X has type at most τ, the type of the value carried by v). Similarly, in rule (D-T-OUT), it is checked that u can access channel v, that message W can be assigned at least type τ in u by Γ, that u can see W and that the region specified for W is at most the region of the values that v can carry.

Environment Extension:

$$\Gamma, {}_u w : \phi \quad \Gamma' \quad \text{s.t. } \Gamma'(v) = \begin{cases} \Gamma(v) & \text{if } v \neq w \text{ and } w \notin dom(\Gamma) \\ \phi & \text{if } v = w \notin dom(\Gamma) \end{cases}$$

$$\Gamma, {}_u w : \gamma \quad \Gamma' \quad \text{s.t. } \Gamma'(v) = \begin{cases} \Gamma(v) & \text{if } v \neq u \\ r \triangleright r'[w : \gamma, \widetilde{w' : \gamma'}]^{p}_{d} & \text{if } v = u \text{ and } w \notin \Theta' \\ & \text{and } \Gamma(u) = r \triangleright r'[\widetilde{w' : \gamma'}]^{p}_{d} \end{cases}$$

$$\Gamma, {}_u x_1@x_2 : \gamma@\phi \quad \Gamma' \quad \text{s.t. } \Gamma'(v) = \begin{cases} \Gamma(v) & \text{if } v \neq x_2 \text{ and } x_2 \notin dom(\Gamma) \\ r \triangleright r'[x_1 : \gamma, u : \gamma]^{p}_{d} & \text{if } v = x_2 \notin dom(\Gamma) \\ & \text{and } \phi = r \triangleright r'[u : \gamma]^{p}_{d} \end{cases}$$

Subtyping Relation:

$$\tau \sqsubseteq \tau \qquad \frac{\tau \sqsubseteq \tau' \qquad \tau' \sqsubseteq \tau}{\tau = \tau'} \qquad \frac{\tau_1 \sqsubseteq \tau_2 \qquad \tau_2 \sqsubseteq \tau_3}{\tau_1 \sqsubseteq \tau_3}$$

$$\frac{r \subseteq s \qquad r' \supseteq s' \qquad r_d \subseteq s_d \qquad r_p \subseteq s_p \qquad n \leq m \qquad \forall i = 1, \cdots, n. \gamma_i \sqsubseteq \gamma'_i}{r \triangleright r'[u_1 : \gamma_1, \ldots, u_n : \gamma_n]^{p}_{d} \sqsubseteq s \triangleright s'[u_1 : \gamma'_1, \ldots, u_m : \gamma'_m]^{p}_{d}}$$

$$\frac{r \subseteq r' \qquad \tau \sqsubseteq \tau'}{r(\tau) \sqsubseteq r'(\tau')} \qquad \frac{\gamma \sqsubseteq \gamma' \qquad \phi \sqsubseteq \phi'}{\gamma@\phi \sqsubseteq \gamma'@\phi'}$$

Table 7: Technicalities of Dπ Typing

Finally, rule (D-T-GO) verifies that u can see v and all the identifiers occurring in P by exploiting function $reg^u_\Gamma(\cdot)$ defined inductively as follows:

$$reg^u_\Gamma(\mathbf{0}) \quad \top \quad reg^u_\Gamma(*P) \quad reg^u_\Gamma(P) \quad reg^u_\Gamma(P_1|P_2) \quad reg^u_\Gamma(P_1) \cap reg^u_\Gamma(P_2)$$
$$reg^u_\Gamma((\nu e)P) \quad reg^u_\Gamma(P) - \{e\} \qquad reg^u_\Gamma(\mathbf{go}\ v.P) \quad reg(\Gamma(v) \cap reg^v_\Gamma(P)$$
$$reg^u_\Gamma(v!\langle [W] \rangle.P) \quad reg(\Gamma(u) \cap v) \cap r \cap reg^u_\Gamma(P)$$
$$reg^u_\Gamma(v?(X).P) \quad reg(\Gamma(u) \cap v) \cap reg^u_\Gamma(P)$$

The premises of the rule also check if u can send P to v (by exploiting the data/process region of v according to the fact that P is an output process or not) and if P typechecks at v.

To conclude, we define well-typed Dπ nets.

Definition 4.11 (Well-typed Dπ Net) *A net N is* well-typed in Γ *if $\Gamma \vdash N$. A net N is* well-typed *if there exists a localization environment Γ such that N is well-typed in Γ.*

4.3.2 Dπ Operational Semantics

Dπ nets evolve according to the reduction relation $\longmapsto$ defined in Table 8. Like in CKLAIM, $\longmapsto$ relates configurations of the form $K \triangleright N$, where K is a set of localities and localised channels (thus, $K = \{l_1, l_2, \ldots, a_1@l_1, a_2@l_2, \ldots\}$) such that $n(N) \subseteq K \subset_{fin} \mathcal{L} \cup (\mathcal{C} \times \mathcal{L})$, and function $n(N)$ returns the set of all (possibly compound) names occurring in N. For example, a suitable K for the net $l^{r_p}_{r_d}[\![a?(X).\mathbf{0}]\!]$ is $\{l, a@l\} \cup r_p \cup r_d$. Like before, $\longmapsto^?$ denotes the reflexive and transitive closure of $\longmapsto$.

Substitutions are now generalised so that they map input parameters to messages and their application keeps into account also the structure of the message/parameter involved. In particular, in $P\{^W/_X\}$, if W is a

(Dπ-COMM)

$$l_{r_d}^{r_p}[\![a?(X).P \mid a!\langle [\![W]\!]\rangle.Q]\!] \longmapsto l_{r_d}^{r_p}[\![P\{^W/_X\} \mid Q]\!]$$

(Dπ-GO)

$$l_{r_d}^{r_p}[\![g\ \ k.P]\!] \| k_{r_d'}^{r_p'}[\![Q]\!] \longmapsto l_{r_d}^{r_p}[\![p\quad]\!] \| k_{r_d'}^{r_p'}[\![P \mid Q]\!]$$

(Dπ-NEWLOC)

$$K \triangleright l_{r_d}^{r_p}[\![(\nu k)P]\!] \longmapsto K \uplus \{k\} \triangleright l_{r_d \cup \{k\}}^{r_p \cup \{k\}}[\![P]\!] \| k_{r_d \cup \{k\}}^{r_p \cup \{k\}}[\![p\quad]\!]$$

(Dπ-NEWCHAN)

$$K \triangleright l_{r_d}^{r_p}[\![(\nu a)P]\!] \longmapsto K \uplus \{a@l\} \triangleright l_{r_d}^{r_p}[\![P]\!]$$

(Dπ-CALL)

$$l_{r_d}^{r_p}[\![*P]\!] \longmapsto l_{r_d}^{r_p}[\![*P \mid P]\!]$$

(Dπ-SPLIT)

$$\frac{K \triangleright l_{r_d}^{r_p}[\![P]\!] \| l_{r_d}^{r_p}[\![Q]\!] \| N \longmapsto K' \triangleright l_{r_d'}^{r_p'}[\![P']\!] \| l_{r_d}^{r_p}[\![Q']\!] \| N'}{K \triangleright l_{r_d}^{r_p}[\![P \mid Q]\!] \| N \longmapsto K' \triangleright l_{r_d'}^{r_p'}[\![P' \mid Q']\!] \| N'}$$

(Dπ-PAR)

$$\frac{K \triangleright N_1 \longmapsto K' \triangleright N_1'}{K \triangleright N_1 \| N_2 \longmapsto K' \triangleright N_1' \| N_2}$$

(Dπ-STRUCT)

$$\frac{N_1 \equiv N_1' \qquad K \triangleright N_1' \longmapsto K' \triangleright N_2' \qquad N_2' \equiv N_2}{K \triangleright N_1 \longmapsto K' \triangleright N_2'}$$

Table 8: Dπ Operational Semantics

name e, then X must be a variable x and the application replaces x with e in P; otherwise, if W is a compound message $a@l$, then X must be a compound variable $z@y$ and the application replaces z with a and y with l in P. Like in CKLAIM, substitution application also acts on the region annotations in P.

The reduction relation relies on a structural congruence relation, $\equiv$, equating α-convertible processes, stating that "$\|$" is commutative and associative, and that $\mathbf{0}$ acts as the identity for "$|$".

We now comment on the Dπ peculiar operational rules; the others are similar to the corresponding ones of CKLAIM. Notice that, differently from CKLAIM, region annotations are not exploited to infer reductions, thanks to the powerful static typing. Rule (Dπ-COMM) states that the producer and the consumer of a datum must locally synchronise along a named channel a. Rule (Dπ-GO) moves the continuation process to the node target of the g; notice that the static typing has already verified that k accepts data/code from l (i.e. $l \in r_d'$ or $l \in r_p'$, according to the fact that P is an output process or not). Rules (Dπ-NEWLOC) and (Dπ-NEWCHAN) handle name restriction. The first one creates a new node addressed by the fresh locality name k; k enlarges the creating node's regions that, similarly to CKLAIM, are assigned to the new node too. The second rule allocates a new channel in the current locality. In both cases, the set K of names already in use is exploited to choose a fresh name. This corresponds to the intuition that, rather than declaring

something as local, one can give it a syntactically different name. The effect is the same as in the standard semantics [34], where structural congruence is exploited to extend the scope of names by moving name restriction to the outermost level.

Before concluding, let us implement in Dπ the example given in Section 4.1. Like in CKLAIM, the addresses are l_C and l_S with region annotations such that $l_S \in r_d^C$ and $l_C \in r_d^S$ (usually, $r_d^S = \top$), while $r_p^C = r_p^S = \varnothing$. Processes P_C and P_S now become

$$P_C \quad (\nu\ \ ).(g\ \ l_S.\quad !\langle\ \ @l_C, [c\ d\ \ itCa\ d\_in\ fo]_{\{l_C;l_S\}}\rangle \mid \ \ ?(y).<ch\ \ ify\ p\ \ \ dyd\ \ a\ \ \ th\ \ \ e\ \ \ >)$$

$$P_S \quad *\quad ?(x_1@x_1',x_2).<ch\ \ k\ c\ d\ \ it\ a\ \ d\ in\ fo\ x_2>.\ (\nu p)(g\ \ x_1'.x_1!\langle [p]_{\{x_1';l_S\}}\rangle \mid <d\ \ \ le\ p\ \ \ \ d\ ch\ \ \ if\ a\ \ t\ o\ \ \ d\ \ \ p\ oi\ \ e\ th\ \ e\ \ \ >)$$

Channel is used as the access point to the server for requiring the service, while is a fresh channel used to transmit secret passwords between the client and the server. For the sake of presentation, we also assume basic values (of type) for passwords and credit card information. Under these assumptions, a possible type environment to typecheck the net

$$l_C{}^{r_p^C}_{r_d^C}[\![P_C]\!] \parallel l_S{}^{r_p^S}_{r_d^S}[\![P_S]\!]$$

is

$$\Gamma\ :\ l_C \mapsto \phi_C \qquad l_S \mapsto \top \triangleright \{l_S\}[\quad : \top(\gamma_{\mathrm{pwd}}@\phi_C \times \ \ )]^{r_p^S}_{r_d^S}$$

where $\phi_C = \top \triangleright \{l_C\}[\,]^{r_p^C}_{r_d^C}$ and $\gamma_{\mathrm{pwd}} = \{l_C, l_S\}(\ \ )$. Clearly, we are not considering the activities of modifying/handling passwords, nor those of accessing/providing the service. A sketch of the type checking for P_C is given below. It requires to establish that

$$\Gamma \vdash_{l_C} (\nu\ \ ).(g\ \ l_S.\quad !\langle\ \ @l_C, [c\ d\ \ itCa\ d\_in\ fo]_{\{l_C;l_S\}}\rangle \mid \ \ ?(y).\cdots)$$

By applying rules (D-T-LRES) and (D-T-PAR), this requires to infer that

$$\Gamma_1 \vdash_{l_C} g\ \ l_S.\quad !\langle\ \ @l_C, [c\ d\ \ itCa\ d\_in\ fo]_{\{l_C;l_S\}}\rangle$$

where $\Gamma_1 = \Gamma, l_C\ \ : \gamma_{\mathrm{pwd}}$. The judgement above can be inferred by using rule (D-T-GO). To this aim, first we establish that $\{l_C\} \subseteq reg_{\Gamma_1}^{l_C}(g\ \ l_S.\quad !\langle\ \ @l_C, [c\ d\ \ itCa\ d\_in\ fo]_{\{l_C;l_S\}}\rangle) = \{l_C, l_S\}$ and $\{l_C\} \subseteq r_d^S$, then we prove that

$$\Gamma_1 \vdash_{l_S}\quad !\langle\ \ @l_C, [c\ d\ \ itCa\ d\_in\ fo]_{\{l_C;l_S\}}\rangle$$

This last judgement is inferred by using rule (D-T-OUT). Indeed, the following conditions are all satisfied:

$$\Gamma_1 \vdash_{l_C}\quad : \top(\gamma_{\mathrm{pwd}}@\phi_C \times \ \ ) \quad \{l_S\} \subseteq \top \quad \Gamma_1 \vdash_{l_C}\ \ @l_C : \gamma_{\mathrm{pwd}}@\phi_C$$
$$\Gamma_1 \vdash_{l_C} c\ d\ \ itCa\ d\_in\ fo : \qquad \Gamma_1 \vdash_{l_S} p$$

R 1 (T Dπ N à la CKLAIM**)** The typing approach used for CKLAIM, where the values exchanged in communications are checked at runtime, can be easily adapted to Dπ. The resulting type system is simpler, e.g. types are simply regions and do not need complex subtyping relations. Of course, the modified setting requires more runtime checks. The definition of well-typed nets, like the type inference phase, can be straightforwardly adapted from Table 2. Clearly, the syntax of Table 5 must be adapted by letting parameters to be defined as

$$X ::= [x] \mid [z]^1@[y]^2$$

As regards the operational semantics, we need to replace rules (Dπ-COMM) and (Dπ-GO) in Table 8 with the following ones:

$$\frac{r' \subseteq r}{l^{r_p}_{r_d}[\![a?(\tilde{x}^{\emptyset}).P \mid a!\langle \tilde{e}^{\emptyset}\rangle.Q]\!] \longmapsto l^{r_p}_{r_d}[\![P\{^{e}/_{x}\} \mid Q]\!]}$$

$$\frac{r_1 \cup r_2 \subseteq r}{l^{r_p}_{r_d}[\![a?(\tilde{z}^{r_1}@\tilde{y}^{r_2}).P \mid a!\langle \tilde{b}@k^{\emptyset}\rangle.Q]\!] \longmapsto l^{r_p}_{r_d}[\![P\{^{b@k}/_{z@y}\} \mid Q]\!]}$$

$$\frac{\text{if } \mathit{out}_p(t(P)) \text{ then } l \in r'_d \text{ else } l \in r'_p}{l^{r_p}_{r_d}[\![g[\,k.P]\!] \,\|\, k^{r'_p}_{r'_d}[\![Q]\!] \longmapsto l^{r_p}_{r_d}[\![\,]\!] \,\|\, k^{r'_p}_{r'_d}[\![P \mid Q]\!]}$$

To conclude the discussion on Dπ, we sketch the proof of its soundness.

Theorem 4.12 (Subject reduction). *If N is well-typed and $K \triangleright N \longmapsto K' \triangleright N'$ then N' is well-typed.*

We now consider type safety. We could proceed like for CKLAIM, by exploiting a tagged language. However, in the Dπ setting, we can formulate and prove the safety property in a simpler (but coarser) way. The intuition is that a Dπ typing environment already associates a region to each (free) name of a net. Thus, we can define a notion of safety *wrt* *a priori* *environment* in the following way:

Definition 4.13 (Safety). *A net N is Γ-safe if for every $l^{r_d}_{r_p}[\![P]\!]$ in N, it holds that $l \in reg^{l}_{\Gamma}(P)$.*

This definition is somehow "less accurate" than Definition 4.5 in that all the occurrences of the same name are now associated with the same tag. To obtain the finer property, we should tune the theory presented in Section 4.2.3; we omit the details and go on working with Definition 4.13.

Theorem 4.14 (Type safety). *If $\Gamma \vdash N$ then N is Γ-safe.*

Type soundness now easily follows (the definition of r-subnet and the proof of the claim are similar to the corresponding ones in Section 4.2.3). Notice that type soundness can be recovered as an instance of *local* type soundness.

Corollary 4.15 (Local type soundness). *Let r be a region for N and $r' = r \cup (K' - K)$ and $\Gamma' = \Gamma$ where $\mathit{dom}(K' - K) \# N$. If $K \triangleright N \longmapsto^{?} K' \triangleright N'$ then r' is a region for N' and Γ' is well-formed.*

4.4 Mobile Ambients Calculus

Finally, we apply our approach to Mobile Ambients Calculus [14] (in the following, we will shorten it as Ambients). The calculus relies on the notion of *ambient* that can be thought of as a bounded place where processes cooperate. This notion is similar to that of node, but differently from CKLAIM and Dπ nodes, ambients can be hierarchically structured and can be moved as a whole under the control of processes.

The syntax of the calculus is given in Table 9. There is only one category of names, namely that of *ambient names*, $\mathcal{A}$, ranged over by n. Identifiers, ranged over by u, v, w, denote ambient names and variables (ranged over by x), and represent both the target of process actions and the data exchanged during communication. Communication is asynchronous and anonymous (no place or communication channel is explicitly referred), and takes place locally within a single ambient.

In Ambients, everything is a process, namely, differently from CKLAIM and Dπ, there is no distinction among processes, nodes and nets. Other than the standard process operators, i.e. empty process, prefixing,

$n, m, p, \ldots \in \mathcal{A}$			AMBIENTS NAMES
$x, y, z, \ldots$			VARIABLES
u	::=	$n \mid x$	IDENTIFIERS
P	::=		PROCESSES
		$\mathbf{0}$	(empty)
	$\mid$	$\langle [u]_r \rangle$	(datum)
	$\mid$	$\mathsf{a}.P$	(prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	(νn)	(name restriction)
	$\mid$	$*P$	(replication)
	$\mid$	$u[P]$	(ambient)
a	::=		ACTIONS
		(x)	(receive)
	$\mid$	**in**_u	(enter u)
	$\mid$	**out**_u	(exit u)
	$\mid$	**open**_u	(open u)

Table 9: Ambients Syntax

parallel composition, name restriction and replication, we have $\langle [u]_r \rangle$, that represents message u tagged with region r within the current ambient, and $n[P]$, that represents an ambient with name n and process P running inside. An ambient, hence, has a name, a collection of local processes and a collection of subambients. Notice that nothing prevents existence of two or more ambients with the same name, possibly enclosed within the same ambient. Process actions are:

- (x): receives a message u within the current ambient and replaces x with u in the continuation;
- **in** _u: moves the ambient enclosing the process executing the action in a sibling ambient whose name is u;
- **out** _u: moves the ambient enclosing the process executing the action out of its enclosing ambient provided that this is named u;
- **open** _u: dissolves the boundary of an ambient named u and unleashes u's content.

Identifiers occurring in processes can be **bound**. More precisely, prefix $(x).P$ binds variable x, while $(\nu n)P$ binds name n; in both cases, P is the scope of the binding. The set of free variables fv(_), α-conversion and closed nets are defined accordingly.

Differently from the calculi previously presented, in Ambients there is no need to associate regions r_d / r_p to ambients. Indeed, as processes are confined within ambients, new data/code can enter an ambient n only because an ambient boundary is dissolved by an action **open** executed within n. Since this action is under the control of n, no (static nor dynamic) check is needed to prevent the **undesired** arrival of undesired data/code. At most, some control can be carried on the ambients that n can open; but this is an orthogonal task.

4.4.1 Typing Ambients Names

We adopt a static typing approach, like for Dπ. Ambients types are defined as follows:

$$T \quad ::= \quad \mathsf{shh} \quad \mid \quad r_1 \triangleright r_2 \triangleright r_3 \, [T]$$

Intuitively, an ambient u has type $r_1 \triangleright r_2 \triangleright r_3 \, [T]$ if its name is in r_3 (this is useful only if u is a variable), it can be seen by all ambients in r_1 and enclosed within all ambients in r_2. Moreover, the ambient hosts processes

$$\frac{}{\Gamma \vdash \mathbf{0}}\ \text{(A-T-EMPTY)} \qquad \frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 | P_2}\ \text{(A-T-PAR)} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash *P}\ \text{(A-T-REPL)}$$

$$\frac{\Gamma \vdash_n P}{\Gamma \vdash n[P]}\ \text{(A-T-AMB)} \qquad \frac{\Gamma, n \mapsto r_1 \triangleright r_2 \triangleright \{n\}\,[T] \vdash P}{\Gamma \vdash (\nu n)P}\ \text{(A-T-RES)}$$

Table 10: Main Judgement for Typing Ambients

exchanging data of type T, the **b** **o** **v** **b** **p** . Topics of conversations were introduced in [13]; here we use them in a similar way and denote with shh the absence of exchanges in the ambient. Moreover, we always assume that types are well-formed, i.e., for all $r_1 \triangleright r_2 \triangleright r_3\,[\cdot]$ it holds that $r_2 \subseteq r_1$. Finally, we let **o** $t(r_1 \triangleright r_2 \triangleright r_3\,[\cdot]) = r_2$ and **lb** $(r_1 \triangleright r_2 \triangleright r_3\,[\cdot]) = r_1$.

The main judgement is $\Gamma \vdash P$ and states that P is well-typed under the assumptions Γ. A **p** **a** **v** **m** Γ is a finite partial function mapping ambient names and variables to types (a well-formedness condition similar to (‡) in Section 4.3.1 is assumed, see below). The key requirement to typecheck Ambients processes is that

> whenever ambient n is contained in ambient m (i.e., $m[n[\cdots] \mid \cdots]$), (§)
> it must hold that **o** $t(\Gamma(m)) \subseteq$ **o** $t(\Gamma(n))$

By construction, we have that **o** $t(\Gamma(n)) \subseteq$ **lb** $(\Gamma(n))$. These conditions together ensure that, if m is opened while still containing n, n can be seen by the ambient enclosing m. For example, consider the following process:

$$k[m[n[\langle d \rangle] \mid \cdots]] \mid$$ **p** $\_n[] \mid$ **p** $\_m[]$

If the process is well-typed, we know that $k \in$ **o** $t(\Gamma(m)) \subseteq \{m\} \cup$ **o** $t(\Gamma(m)) \subseteq$ **o** $t(\Gamma(n)) \subseteq r$ (these inclusions follow by the premises of the typing rules). This means that both k and m can see datum d; hence, the execution of actions **p** $\_n$ and **p** $\_m$ does not break well-typedness, as intended.

The typing procedure for Ambients processes is presented in Tables 10 and 11. It is somehow inspired by the basic typing of [13] and also includes some features we have already presented for Dπ in Section 4.3.1.

The main judgement $\Gamma \vdash P$ is defined by the rules in Table 10. Intuitively, it states that P is well-typed under the assumptions Γ. Differently from [14], we do not assign a type to processes and consider ill-typed those processes with actions or messages occurring outside any ambient boundary. The latter choice reflects our intuition of ambients as nodes of a net: a process cannot perform any computational activity if it has not been allocated within some ambient. The main judgement relies on the auxiliary judgement $\Gamma \vdash_u P$ defined by the rules in Table 11. This judgement is invoked in rule (A-T-AMB) of the main judgement and states that, when located within ambient u, process P can be typed in the environment Γ. As a matter of notation, $\Gamma, u \mapsto T$ will stand for the type environment Γ' such that $\Gamma'(v) = \Gamma(v)$, if $v \neq u$ and $u \notin$ **dom** (Γ), and $\Gamma'(u) = T$ if $u \notin$ **dom** (Γ).

We use functions **lb** (T), **o** $t(T)$ and **ub** (T) to denote the regions r_1, r_2 and r_3, respectively, when $T = r_1 \triangleright r_2 \triangleright r_3\,[\cdot]$; **lb** (shh) denotes $\top$. Finally, we also assume the following well-formedness condition on environments:

> Let $\Gamma(u) = r_1 \triangleright r_2 \triangleright r_3\,[T]$. For every $n \in r_2$ it must be that **o** $t(\Gamma(n)) \subseteq r_2$. Moreover, if $u \in \mathcal{A}$ then $r_3 = \{u\}$; otherwise, for each $n \in r_3$, it must be that $r_1 \subseteq$ **lb** $(\Gamma(n))$ and $r_2 \subseteq$ **o** $t(\Gamma(n))$.

We now briefly comment on some key features of the type system. For each occurrence of an identifier u, it is verified that the ambient containing the occurrence, and all the possibly enclosing ambients, can

(AA-T-NIL)
$$\frac{}{\Gamma \vdash_u \mathbf{0}}$$

(AA-T-PAR)
$$\frac{\Gamma \vdash_u P_1 \qquad \Gamma \vdash_u P_2}{\Gamma \vdash_u P_1 | P_2}$$

(AA-T-REPL)
$$\frac{\Gamma \vdash_u P}{\Gamma \vdash_u *P}$$

(AA-T-AMB)
$$\frac{\mathit{lub}\ (\Gamma(u) \cup \mathit{co}\ t(\Gamma(u)) \subseteq \mathit{co}\ t(\Gamma(v)) \quad \Gamma \vdash_v P}{\Gamma \vdash_u v[\![P]\!]}$$

(AA-T-RES)
$$\frac{\Gamma, n \mapsto r_1 \triangleright r_2 \triangleright \{n\}[T] \vdash_u P}{\Gamma \vdash_u (\nu n)P}$$

(AA-T-IN)
$$\frac{\mathit{lub}\ (\Gamma(v) \cup \mathit{co}\ t(\Gamma(v)) \subseteq \mathit{lbo}\ (\Gamma(u)) \qquad \mathit{lub}\ (\Gamma(u) \cup \mathit{co}\ t(\Gamma(u)) \subseteq \mathit{co}\ t(\Gamma(v)) \qquad \Gamma \vdash_v P}{\Gamma \vdash_v \mathbf{in}\ \_u.P}$$

(AA-T-OUT)
$$\frac{\mathit{lub}\ (\Gamma(v) \cup \mathit{co}\ t(\Gamma(v)) \subseteq \mathit{lbo}\ (\Gamma(u)) \qquad \Gamma \vdash_v P}{\Gamma \vdash_v \mathbf{out}\ \_u.P}$$

(AA-T-OPEN)
$$\frac{\mathit{lub}\ (\Gamma(v) \cup \mathit{co}\ t(\Gamma(v)) \subseteq \mathit{lbo}\ (u) \qquad \Gamma \vdash_v P}{\Gamma \vdash_v \mathbf{open}\ \_u.P}$$

(AA-T-RCV)
$$\frac{\Gamma(u) = r_1 \triangleright r_2 \triangleright r_3[T] \qquad \Gamma, x \mapsto T \vdash_u P}{\Gamma \vdash_u (x).P}$$

(AA-T-SND)
$$\frac{\Gamma(v) = r_1 \triangleright r_2 \triangleright r_3[T] \quad \Gamma(u) = T \quad \mathit{lub}\ (\Gamma(v) \cup \mathit{co}\ t(\Gamma(v)) \subseteq \bigsqcap_{w \in r} \mathit{lub}\ (\Gamma(w)) \subseteq \mathit{lbo}\ (T))}{\Gamma \vdash_v \langle [u] \rangle}$$

Table 11: Auxiliary Judgement for Typing Ambients

see u (see rules (AA-T-IN), (AA-T-OUT), (AA-T-OPEN) and (AA-T-SND) – and, indirectly, also rule (AA-T-AMB)). The crucial rules are (AA-T-AMB) and (AA-T-IN): they ensure that the ambient hierarchy always maintains the invariant (§) described in Section 4.4.1. Finally, rules (AA-T-RCV) and (AA-T-SND) exploit the topic of conversation to handle communications, i.e. to assign types to input variables or to verify that messages are sent at the type required by the ambient where the exchanges take place.

Weak for Ambients processes is defined as follows.

Definition 4.16 (Weak Subject Reduction) *A process P is* well-typed in Γ *if* $\Gamma \vdash P$ *P is* well-typed *if there exists an environment Γ such that P is well-typed in Γ.*

Remark 2 To conclude this section, we want to remark that most of the intricacies in the setting of the Ambients calculus (especially, the need for requirement (§) above) are related to the presence of action **open**. Indeed, other calculi, that have been derived from Ambients by removing such a primitive, can be typed very similarly to Dπ. As a first example, we consider M^3 [17] where action **open** is replaced by a primitive for process migration, **go**, which is in the same vein of CKLAIM's **eval** and Dπ's **go**. In this setting, types look like Dπ's locality types and are defined as $r \triangleright r'[T]^{p}_{d}$ (the meaning of r, r', r_d and r_p is like in Dπ, while T is the topic of conversation). As another example, we consider Boxed Ambients [6] where action **open** is replaced with primitives for (non local) parent/child communication. Types still take the form $r \triangleright r'[T]^{p}_{d}$, but more checks are needed in the typing phase to ascertain that data are exchanged correctly.

(A-IN)

$$\overline{n[\text{in } \_m.P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]}$$

(A-OUT)

$$\overline{m[n[\text{out } \_m.P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]}$$

(A-OPEN)

$$\overline{\text{open } \_m.P \mid m[R] \longrightarrow P \mid R}$$

(A-COMM)

$$\overline{\langle m \rangle \mid (x).Q \longrightarrow Q\{^m/_x\}}$$

(A-STRUCT)

$$\frac{P_1 \equiv P_1' \qquad A \triangleright P_1' \longrightarrow A' \triangleright P_2' \qquad P_2' \equiv P_2}{A \triangleright P_1 \longrightarrow A' \triangleright P_2}$$

(A-CALL)

$$\overline{*P \longrightarrow *P \mid P}$$

(A-RES)

$$\overline{A \triangleright (\nu n)P \longrightarrow A \uplus \{n\} \triangleright P}$$

(A-PAR)

$$\frac{A \triangleright P_1 \longrightarrow A' \triangleright P_1'}{A \triangleright P_1 \mid P_2 \longrightarrow A' \triangleright P_1' \mid P_2}$$

(A-AMB)

$$\frac{A \triangleright P \longrightarrow A' \triangleright P'}{A \triangleright n[P] \longrightarrow A' \triangleright n[P']}$$

Table 12: Ambients Operational Semantics

Indeed, the ability of performing (limited forms of) remote communications introduces new possibilities to forge data regions.

4.4.2 [illegible]

Ambients processes are executed according to the reduction relation $\longrightarrow$ defined in Table 12. Like for the previous calculi, $\longrightarrow$ relates configurations of the form $A \triangleright P$, where A is now a set of ambient names such that $n(P) \subseteq A \subset_{fin} \mathcal{A}$. Function $n(P)$ returns the set of ambient names occurring in P. As usual, $\longrightarrow^?$ stands for the reflexive and transitive closure of $\longrightarrow$.

The semantics is given modulo a [illegible] relation, $\equiv$, equating α-convertible processes and stating that "|" is commutative, associative and with $\mathbf{0}$ as identity.

We now comment on the Ambients peculiar operational rules. Rule (A-IN) says that the ambient n performing the action enters the sibling ambient m. If no sibling m can be found, the operation gets stuck until such a sibling exists; if more than one sibling m exists, any one of them can be chosen. Symmetrically, rule (A-OUT) says that the ambient n performing the action exits its enclosing ambient if this is named m; otherwise, the action gets stuck. Rule (A-OPEN) says that the boundary of ambient n is dissolved and n's content is unleashed, possibly within the ambient performing the action. If no ambient n is found, the operation gets stuck until such an ambient exists; if more than one ambient n exists, any one of them can be chosen. Rule (A-COMM) accounts for asynchronous communication between co-located processes; again, the static typing enables the communication without any runtime overhead. In rule (A-RES), the set A of names already in use is exploited to choose a fresh name n. Finally, rule (A-AMB) states that, if the content of an ambient evolves, then the whole ambient evolves accordingly.

Before concluding, let us implement in Ambients the example presented in Section 4.1. The server and the client are modelled as two sibling ambients whose names are n_S and n_C. Processes P_C and P_S now

become

$$
\begin{array}{lll}
P_C & \triangleq & (\nu\ \tilde{v})\ (\ \ |[\mathbf{out}\ n_C.\mathbf{in}\ n_S.\langle n_C,\ \tilde{v}\ ,[c\ !\dot{n}\ fo]_{\{n_C;n_S;\mathsf{req}\}}\rangle]] \\
 & & \quad |\ \mathbf{open}\ \tilde{v}\ .(y).<\mathit{dh}\ \ \mathit{if}\ y\ \mathit{pdv}\ \ y\ \mathit{d}\ \ \mathit{ea}\ \ \ \mathit{tha}\ \ \mathit{e}\ \ \dot{v}\mathit{e}\ \ >) \\
P_S & \triangleq & *\mathbf{open}\ \ .(x_1,x_2,x_3).<\mathit{cha}\ \ k\ \mathit{c}\ \mathit{d}\ \ \mathit{it}\ \mathit{a}\ \ \mathit{d}\ \dot{n}\ \mathit{fo}\ x_3>. \\
 & & (\nu p)(\ x_2[[\mathbf{out}\ \ .n_S.\mathbf{in}\ \ .x_1.\langle[p]_{\{x_1;x_2;n_S\}}\rangle]]\ | \\
 & & \quad <\mathit{d}\ \ \mathit{le}\ \mathit{pv}\ \ \ \mathit{d}\ \mathit{dh}\ \ \mathit{if}\ \mathit{a}\ \ \mathit{tis}\ \mathit{o}\ \ \ \mathit{d}\ \ \ \mathit{p}\ \mathit{oid}\ \ \mathit{e}\ \mathit{tha}\ \ \mathit{e}\ \ \dot{v}\mathit{e}\ \ >)
\end{array}
$$

Ambient is used as an access point to the server; indeed, it brings message $\langle n_C,\ \tilde{v}\ ,[c\ !\dot{n}\ fo]_{\{n_C;n_S;\mathsf{req}\}}\rangle$ out of the client and then in the server, where it is dissolved, thus enabling the reception of the message. This is carried on by the following reductions (where, like before, the sets A of names in use are omitted):

$$
\begin{array}{l}
n_C[[P_C]]\ |\ [[n_S]]P_S \\
\longrightarrow^{?} n_C[[\mathbf{open}\ \ \tilde{v}\ .\cdots]]\ |\ n_S[[P_S]]\ |\ \ \ [[\mathbf{in}\ \ n_S.\langle n_C,\ \tilde{v}\ ,[c\ !\dot{n}\ fo]_{\{n_C;n_S;\mathsf{req}\}}\rangle]] \\
\longrightarrow n_C[[\mathbf{open}\ \ \tilde{v}\ .\cdots]]\ |\ n_S[[P_S\ |\ \ \ [[\langle n_C,\ \tilde{v}\ ,[c\ !\dot{n}\ fo]_{\{n_C;n_S;\mathsf{req}\}}\rangle]]]] \\
\longrightarrow n_C[[\mathbf{open}\ \ \tilde{v}\ .\cdots]]\ |\ n_S[[P_S\ |\ \langle n_C,\ \tilde{v}\ ,[c\ !\dot{n}\ fo]_{\{n_C;n_S;\mathsf{req}\}}\rangle\ |\ (x_1,x_2,x_3).\cdots]]
\end{array}
$$

Upon verification of the credit card information, the server creates a new password p that is delivered back to n_C by the ambient $\tilde{v}$. Again, this last ambient serves as an access point to the client and acts like ambient above.

By comparing this implementation with that in Dπ, one can notice how channels can be implemented in Ambients. Intuitively, channels (e.g., and $\tilde{v}$ in the example above) are rendered as pilot ambients that bring messages from the sender to the receiver, by following possibly complex routing paths. Once they have reached their final destination, such ambients are opened so that the messages they carry on are unleashed and can be retrieved by the receiver.

To conclude, we now sketch the subject reduction and type safety theorems for Ambients. to this aim, we give a Lemma (crucial in the case for) that formally justifies the invariant (§) given in Section 4.4.1.

Lemma 4.17 *If* $\Gamma \vdash_v P$ *and* $(\Gamma(u) \subseteq a\ \ t(\Gamma(v)$, *then* $\Gamma \vdash_u P$

Theorem 4.18 (Subject Reduction) *Let* $A \triangleright P \longrightarrow A' \triangleright P'$. *If* P *is well typed then* P' *is well typed*

We now formulate and prove type safety and type soundness by following the guidelines of Section 4.3.2. We exploit function $reg_\Gamma(\cdot)$ that is defined inductively as follows:

$$
\begin{array}{c}
reg_\Gamma(\mathbf{0})\ \ \varnothing \qquad reg_\Gamma(P|Q)\ \ reg_\Gamma(P) \cap reg_\Gamma(Q) \qquad reg_\Gamma(u[[P]])\ \ \mathit{lo}\ \ (\Gamma(u) \\
reg_\Gamma(\mathbf{in}\ \ u.P) = reg_\Gamma(\mathbf{out}\ \ u.P) = reg_\Gamma(\mathbf{open}\ \ u.P)\ \ \mathit{lo}\ \ (\Gamma(u) \cap reg_\Gamma(P) \\
reg_\Gamma(\langle[u]\rangle)\ \ \bigcup_{w \in r}\ \mathit{lo}\ \ (\Gamma(w) \qquad reg_\Gamma(*P) = reg_\Gamma((x).P)\ \ reg_\Gamma(P)
\end{array}
$$

Lemma 4.19 (Safety) *Assume* P *is* Γ-safe *for* Γ *and* $u[[Q]]$ *for* P *it holds that* $(\Gamma(u) \subseteq reg_\Gamma(Q)$.

Theorem 4.20 (Type Safety) *If* $\Gamma \vdash P$ *then* P *is* Γ*-safe*.

Like in Dπ, a localised formulation of type soundness can be given. Indeed, we can define the Γ-r-subprocess of P as the process containing all the subambients $n[[Q]]$ of P such that $\{n\} \cup \{m : n \in a\ \ t(\Gamma(m)\ \} \cup a\ \ t(\Gamma(n) \subseteq r$. Thus, $n[[Q]]$ is part of the Γ-r-subprocess of P if and only if r contains n, all the ambients that n can contain and all the ambients that can contain n. This amounts to say that at least all the subtree rooted in n and all the ancestors of n in the ambient hierarchy must be typed to ensure the safety of data occurring in n. Again, since the Γ-$\top$-subprocess of P is P itself, the following theorem also states the soundness result when P is fully typed.

Theorem 4.21 (Subject Reduction [illegible]). Let $\Gamma \vdash_r P$ [illegible] P [illegible] in Γ. If $A \triangleright P \longrightarrow^{?} A' \triangleright P'$, then $\Gamma' \vdash_{r'} P'$ is [illegible], where $r' = r \cup (A' - A)$ and $\Gamma' = \Gamma$ [illegible] for $A' - A$ [illegible]$'$.

4.5 A Realistic Example: Implementing a Multiuser System

In this section we want to further illustrate our approach. To this aim, we use the framework presented so far to program a simple but meaningful example in cKLAIM; the implementations in the other calculi can be derived straightforwardly. For the sake of readability, we will use parameterised process definitions and strings. Moreover, we borrow from [28] polyadic communication, i.e. the possibility of exchanging tuples of data, and the primitive **read** that behaves similarly to **in** but, after its execution, it leaves the accessed data in the TS. Clearly, the type inference for actions **read** works similarly to that for actions **in** (by adding region annotations to parameters occurring in templates).

We present the behaviour of a simple UNIX-like multiuser system, where users can login (exploiting a password-based approach) and use the system functionalities, which consist in reading/writing files or executing programs. For the sake of presentation, we shall present the system in three steps and, finally, we shall merge them together. Let l_S be the address of the server, $\top$ be its data trust region and $\varnothing$ be its process trust region (thus no user can spawn code to l_S).

User identification. We start with programming the identification of different users via passwords. Localities play the role of user IDs. Let l_p be a private repository used by l_S to record the registered users and their passwords. Thus, l_p hosts the tuples

$$\langle l_1, [pwd_1]_{\{l_1;l_p;l_S\}}\rangle \;\;|\;\; \dots \;\;|\;\; \langle l_n, [pwd_n]_{\{l_n;l_p;l_S\}}\rangle$$

Let l be a user wanting to log in l_S. If l is already known to l_S (i.e. it is one of the l_is), then l can use a process like

$$\mathbf{out}(\text{“}login\text{”}, l, [pwd]_{\{l;l_S\}})@l_S.\mathbf{in}(\text{“}logged\text{”})@l_S.\dots$$

for communicating with the server process

$$Login(l_p) \triangleq *\,\mathbf{in}(\text{“}login\text{”}, !u, !z)@l_S.\mathbf{read}(u, z)@l_p.\mathbf{out}([\text{“}logged\text{”}]_{\{l_S;u\}})@l_S$$

Intuitively, l requires a connection by sending its user ID (its locality) and its password; the server checks if this information is correct and sends back an ack, activating the continuation of the computation at l. Notice that the region annotations of pwd and “$logged$” rule out attacks of a nasty intruder aimed at cancelling the request of login or the corresponding ack, and preserve the secrecy of the password.

If the user is not registered at l_S yet, he can send an “hello" request to the server containing its address and wait for a password

$$\mathbf{out}([\text{“}hello\text{”}]_{\{l;l_S\}}, l)@l_S.\mathbf{in}(\text{“}registered\text{”}, !pwd)@l_S.\dots$$

The server then handles this request with the process

$$\begin{aligned} NewUser(l_p) \triangleq\; & *\,\mathbf{in}(\text{“}hello\text{”}, !u)@l_S.\mathbf{new}(pwd).\mathbf{out}(u, [pwd]_{\{u;l_S;l_p\}})@l_p. \\ & \mathbf{out}(\text{“}registered\text{”}, [pwd]_{\{l_S;u\}})@l_S \end{aligned}$$

Of course, a locality l' different from l can send l_S a request for a new password pretending to be l: the only difference with the “hello" message given above is that the message now should contain also l' in the data region. However, the server will report the new password to l and the region associated to the password will ensure that pwd will not leave l. Thus, l' cannot withdraw pwd: it can only try to send a process to l for acting at l with the new password. This can be possible only if l trusts l', implying that l accepts this ‘suspicious’ activity of l'.

We now show the use of our typing theory in the setting just presented. In particular, we give evidence of how we can prevent attacks aimed at cancelling messages and activities of malicious users pretending to play the role of other users.

- Let l_{canc} be a locality hosting a process that aims at interfering with the login procedure above by performing action $\mathbf{in}\,(\text{“}\mathbf{lo}\text{”}, !x)@l_S$. In this way, it removes the $\mathbf{lo}$ message sent by an unregistered user l willing to be connected with the server l_S. The system is modelled as follows

$$\begin{array}{l} l_{\text{canc}} :: \mathbf{in}\,(\text{“}\mathbf{lo}\text{”}, !x)@l_S.\mathbf{0} \quad E \parallel l_S :: \mathit{NewUser}\,(l_p) \\ \parallel l :: \mathbf{out}\,(\llbracket \text{“}\mathbf{lo}\text{”} \rrbracket_{\{l;l_S\}}, l)@l_S.\mathbf{in}\,(\text{“}\mathit{register\ me}\text{”}, !pw)@l_S.\cdots \\ \quad \succ\!\!\rightarrow \quad l_{\text{canc}} :: \mathbf{in}\,(\text{“}\mathbf{lo}\text{”}, !x)@l_S.\mathbf{0} \quad E \\ \qquad \parallel l :: \mathbf{in}\,(\text{“}\mathit{register\ me}\text{”}, !pw)@l_S.\cdots \\ \qquad \parallel l_S :: \mathit{NewUser}\,(l_p) \mid \langle \llbracket \text{“}\mathbf{lo}\text{”} \rrbracket_{\{l;l_S\}}, l \rangle \\ \quad \succ\!\!\not\rightarrow \quad l_{\text{canc}} :: \mathbf{0} \quad E \parallel l :: \mathbf{in}\,(\text{“}\mathit{register\ me}\text{”}, !pw)@l_S.\cdots \\ \qquad \parallel l_S :: \mathit{NewUser}\,(l_p) \end{array}$$

  Notice that the last transition cannot take place. As expected, the intruder running at l_{canc} is not enabled to withdraw the tuple $\langle \llbracket \text{“}\mathbf{lo}\text{”} \rrbracket_{\{l;l_S\}}, l \rangle$ because $l_{\text{canc}} \notin \{l, l_S\}$ (see the runtime check of rule (CK-MATCH)).

- Let now l_{pret} be a locality pretending to act on behalf of l, by trying to acquire a log to l_S under the identity of l. Let us examine the possible evolutions of the system:

$$\begin{array}{l} l_{\text{pret}} :: \mathbf{out}\,(\text{“}\mathbf{lo}\text{”}, l)@l_S.\mathbf{in}\,(\text{“}\mathit{register\ me}\text{”}, !pw)@l_S.\mathbf{0} \quad E \\ \parallel l_S :: \mathit{NewUser}\,(l_p) \\ \quad \succ\!\!\rightarrow \succ\!\!\rightarrow \succ\!\!\rightarrow \succ\!\!\rightarrow \quad l_{\text{pret}} :: \mathbf{in}\,(\text{“}\mathit{register\ me}\text{”}, !pw)@l_S.\mathbf{0} \quad E \\ \qquad\qquad\qquad\qquad \parallel l_S :: \mathit{NewUser}\,(l_p) \mid \langle \text{“}\mathit{register\ me}\text{”}, \llbracket pw \rrbracket_{\{l_S;l\}} \rangle \\ \quad \succ\!\!\not\rightarrow \quad l_{\text{pret}} :: \mathbf{0} \quad E \parallel l_S :: \mathit{NewUser}\,(l_p) \end{array}$$

  Again, the last reduction cannot take place because $l_{\text{pret}} \notin \{l_S, l\}$. The only way for l_{pret} to withdraw the tuple $\langle \text{“}\mathit{register\ me}\text{”}, \llbracket pw \rrbracket_{\{l_S;l\}} \rangle$ is to spawn a process to l (if it exists in the net) executing action $\mathbf{in}\,(\text{“}\mathit{register\ me}\text{”}, !pw)@l_S$ (that would be enabled, because $l \in \{l_S, l\}$). Such a migration, however, should be authorised by l (indeed, it can take place only if $l_{\text{pret}} \in r_p^l$, where r_p^l is the node region controlling migrations to l).

The File System We now consider a server handling a file system where different users can write/read data. Let l_f be a private repository used by l_S to store the files. A file named N, whose content is the string S, that can be read by users in r and written by users in r', is stored in l_f as the process

$$C_N \quad \langle N, \llbracket \text{“}\mathit{read}\text{”} \rrbracket_{r \cup \{l_S;l_f\}}, \llbracket \text{“}\mathit{write}\text{”} \rrbracket_{r' \cup \{l_S;l_f\}} \rangle \quad \mid \quad \langle N, S \rangle$$

Intuitively, “*read*" and “*write*" are just dummy data used to properly store regions r and r'. Then, the server handles requests for reading and writing files with the following processes

$$\begin{array}{ll} \mathit{Read}(l_f) & *\mathbf{in}\,(\text{“}\mathit{read}\text{”}, !u, !n)@l_S.\mathbf{read}\,(n, !z_r, !z_w)@l_f.\mathbf{read}\,(n, !z)@l_f. \\ & \quad \mathbf{out}\,(\llbracket z_r \rrbracket_{\{l_f;l_S;u\}}, n, z)@u \\ \mathit{Write}(l_f) & *\mathbf{in}\,(\text{“}\mathit{write}\text{”}, !u, !n, !z)@l_S.\mathbf{read}\,(n, !z_r, !z_w)@l_f.\mathbf{in}\,(n, !z')@l_f. \\ & \quad \mathbf{out}\,(n, z)@l_f.\mathbf{out}\,(\llbracket z_w \rrbracket_{\{u;l_f;l_S\}}, n)@u \end{array}$$

Intuitively, the first $\mathbf{in}$ action collects the request for reading/writing the file named n performed by locality u; then the following $\mathbf{read}$ action, once type checked, verifies if the locality replacing u has the read/write privilege on file n (see below). Finally, the required operation is performed (the content of the file is read

or the old content is replaced with the new one) and an acknowledgement (containing the kind of operation performed, the name of the file and, in the "read" case, also its content) is sent back to u.

We now show how our types can control read accesses to files. There are two features devoted to this aim: the type inference phase carried on process $\mathit{TAK}(l_f)$ and the runtime checks of the operational semantics. We first give the type inference; recall that absence of region annotations stands for $\top$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{l_S,u\}\subseteq s \qquad \Gamma_1\vdash \mathbf{h}\ \ \succ_{l_S}\ \Gamma_1\vdash \mathbf{h}}{\Gamma_1\vdash \mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u\ \ \succ_{l_S}\ \ \Gamma_2\vdash \mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u}\\ \Gamma_3\vdash \mathbf{rcd}\ (n,!z)@l_f.\mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u\ \succ_{l_S}\\ \Gamma_4\vdash \mathbf{rcd}\ (n,!z)@l_f.\mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u}\\ \Gamma_5\vdash \mathbf{rcd}\ (n,!z_r,!z_w)@l_f.\mathbf{rcd}\ (n,!z)@l_f.\mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u\ \succ_{l_S}\\ \Gamma_6\vdash \mathbf{rcd}\ (n,\lfloor z_r\rfloor^{\{l_S\}},\lfloor z_w\rfloor^{\{l_S\}})@l_f.\mathbf{rcd}\ (n,!z)@l_f.\mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u}\\ \varnothing\vdash \mathbf{in}\ (\text{"rcd"},!u,!n)@l_S.\mathbf{rcd}\ (n,!z_r,!z_w)@l_f.\\ \quad \mathbf{rcd}\ (n,!z)@l_f.\mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u\\ \succ_{l_S}\ \varnothing\vdash \mathbf{in}\ (\text{"rcd"},\lfloor u\rfloor^{\{l_S\}},!n)@l_S.\mathbf{rcd}\ (n,\lfloor z_r\rfloor^{\{l_S\}},\lfloor z_w\rfloor^{\{l_S\}})@l_f.\\ \quad \mathbf{rcd}\ (n,!z)@l_f.\mathbf{out}\ (\lfloor z_r\rfloor_s,n,z)@u$$

where we let $s = \{l_f,l_S,u\}$ and

$$\begin{array}{llll}\Gamma_1 = \Gamma_3\uplus\{z:\{l_S\}\} & & \Gamma_2 = \Gamma_1+\{z_r:s,n:\top,z:\top\}\\ \Gamma_3 = \Gamma_5\uplus\{z_r:\{l_S\},z_w:\{l_S\}\} & & \Gamma_4 = \{u:\{l_S\},n:\top,z_r:s,z_w:\{l_S\}\}\\ \Gamma_5 = u:\{l_S\},n:\{l_S\} & & \Gamma_6 = u:\{l_S\},n:\top\end{array}$$

We now call $\mathit{TAK}(l_f)$ the process obtained from the typing inference above. Let l be a user wanting to read a file named FILE associated to a read region $\rho=\{l_f,l_S,l,\ldots\}$. FILE is then stored at l_f as the process

$$C_{\mathit{FILE}} = \langle \mathit{FILE}, \lfloor\text{"rcd"}\rfloor_s, \lfloor\text{"write attempt"}\rfloor_\emptyset\rangle \mid \langle \mathit{FILE}, \mathit{out}\ \mathit{text}\rangle$$

The evolution of user l is

$$\begin{array}{l} l::\mathbf{out}\ (\text{"rcd"},l,\mathit{FILE})@l_S.\mathbf{in}\ (\text{"rcd"},\mathit{FILE},!\mathit{out}\ \mathit{text})@l.P\\ \parallel l_f::C_{\mathit{FILE}} \parallel l_S::\mathit{TAK}(l_f)\\ \quad\succ\!\!\!\rightarrow\ \succ\!\!\!\rightarrow\ \ l::\mathbf{in}\ (\text{"rcd"},\mathit{FILE},!\mathit{out}\ \mathit{text})@l.P \parallel l_f::C_{\mathit{FILE}}\\ \qquad\qquad \parallel l_S::\mathit{TAK}(l_f) \mid \mathbf{rcd}\ (\mathit{FILE},\lfloor z_r\rfloor^{\{l_f,l_S,l\}},\lfloor z_w\rfloor^{\{l_S\}})@l_f.\cdots\end{array}$$

Now, action $\mathbf{rcd}\ (\mathit{FILE},\lfloor z_r\rfloor^{\{l_f,l_S,l\}},\lfloor z_w\rfloor^{\{l_S\}})@l_f$ is enabled, because $\{l_f,l_S,l\}\subseteq\rho$; thus, the content of FILE will be transferred to l that, in turn, will be enabled to retrieve it (by binding $\mathit{out}\ \mathit{text}$ to the variable $\mathit{out}\ t$) and use it in P. Notice that, if a user $l'\notin\rho$ had tried to carry on the same task, these actions would not have been enabled, since $\{l_f,l_S,l'\}\not\subseteq\rho$.

Exercise (Downloading code). In this last setting, a user can dynamically download some code from the server to perform a given task. The server stores all the downloadable processes as executable (named) files in a private locality l_c. For each executable file named N, whose code is P and that is downloadable by nodes in r, the server stores in l_c the component

$$\begin{array}{rl} C_N = & \langle N,\lfloor\text{"download"}\rfloor_{r\cup\{l_S,l_c\}}\rangle \mid\\ & *\mathbf{in}\ (\mathit{req},N,!u)@l_c.\mathbf{rcd}\ (N,!z_e)@l_c.\\ & \quad \mathbf{eval}\ (\mathbf{out}\ (\lfloor z_e\rfloor_{\{l_c,l_S,u\}},N)@u.P)@u)@l_S\end{array}$$

Then, when a user wants to download some code, the server handles its request with the process

$$\mathit{Exec}(l_c) = *\mathbf{in}\ (\text{"execute"},!u,!n)@l_S.\mathbf{out}\ (\mathit{req},n,u)@l_c$$

Notice that l_c cannot directly send P for execution to u because (the locality associated to) u cannot have l_c in its trust region (since l_c is fresh). Thus, P must firstly cross l_S and then, if l_S is in the process trust region of u (which we assume it is the case), the code-on-demand procedure successfully terminates, by also reporting an ack to the user.

The System Finally, we can put together the activities shown so far to obtain the implementation of the whole server. Thus, the (not yet typed) initial configuration of l_S would be

$$
\begin{array}{l}
l_S\ \top :: \varnothing\ \ \mathbf{new}\ (u_1).\mathbf{new}\ (u_2).\mathbf{new}\ (u_3).\\
\quad < \mathit{setup}\ u_1\ \mathit{with\ the\ identities\ and\ passwords\ of\ the\ users} > .\\
\quad < \mathit{setup}\ u_2\ \mathit{with\ the\ data\ of\ the\ files\ system} > .\\
\quad < \mathit{setup}\ u_3\ \mathit{with\ the\ new\ and\ old\ passwords} > .\\
\quad (\mathit{NewUser}(u_1) \mid \mathit{login}(u_1) \mid \mathit{Read}(u_2) \mid\\
\quad\ \mathit{Write}(u_2) \mid \mathit{Execute}(u_3))
\end{array}
$$

Our example simplifies UNIX behaviour in two major aspects. Firstly, we did not require that a user must login before using the functionalities offered by the system; secondly, the files/programs are put by the system and not by the users. Both these choices were driven by the aim of simplifying the presentation; however, our setting could be easily enriched with more refined and realistic features.

Finally, we want to remark that, by exploiting the dummy data "*read*", "*write*" and "*new/old*", we have been able to enforce an access control policy by only using region annotations. This confirms that, in spite of its simplicity, the approach we presented in this section is very powerful.

4.6 Concluding Remarks

The main contribution of this section is the introduction of a typing discipline for fixing the network region where data and processes can safely move. Our types can prevent execution of those actions that could compromise region specifications. To provide evidence of the generality of our approach, it has been applied to three paradigmatic calculi for global computing with quite different design choices. The technique developed works even when only a local knowledge of the net during the compilation can be assumed and misbehaving entities are present in the system. A few example applications implemented in all the three calculi have been also shown.

We want to remark that our theory permits to naturally implement a security mechanism based on *old/new*. We already noticed, in the introduction, that nodes can be seen as logical partitions of a single physical machine. By exploiting this intuition, one can split each machine into an appropriate number of nodes each with its own security policy. In this way, fine grained security policies can be programmed to guarantee that untrusted processes (e.g., coming from unchecked nodes) are accepted only at dedicated nodes and that from these nodes remote operations and spawning of threads are not permitted.

We conclude by commenting on the three instantiations of our region based approach; this should also shed light on the different choices and paradigms underlying the considered calculi. We shall also discuss possible extensions and confront our work with that of other researchers.

Dynamic Checks Differently from that of Dπ and Ambients, the typing of CKLAIM requires some dynamic checks; this feature, however, is orthogonal to data secrecy and only depends on the underlying process calculus. In fact, these runtime checks could be avoided, and a typing discipline similar to that of Ambients could be developed for CKLAIM too, by additionally requiring that all data exchanged within a node have the same type. However, we find it too demanding to force a CKLAIM node to contain only data of the same type. On the contrary, Dπ channels can be reasonably assigned a fixed type because they can be seen as methods a node supplies to the processes it hosts, and ambients can be reasonably assigned the same type because, due to their hierarchical organisation, they could be thought of as logical partitions of the same memory space.

There is a thread-off between simplicity, efficiency and implementability. The type system for CKLAIM is quite simple and easily implementable (types are just sets and operations on types are unions, intersections, subset inclusions, ...). Clearly, its runtime semantics is less efficient because of the dynamic checks in rules (CK-OUT), (CK-EVAL), (CK-IN) and (CK-MATCH). Nevertheless, we consider reasonable this dynamic burden, since it only involves efficiently implementable operations on sets. On the contrary, Dπ and Ambients have very efficient runtime semantics (no type related check is present) at the expense of more involved static semantics. Also implementability is not straightforward: a preliminary implementation for the (simpler) type system of [34] appeared in a very technical paper [38]; type inference algorithms for Ambients-like calculi are even more complex (see, e.g., [17, 27, 51]). Since our type systems for Dπ and Ambient do rely on [34] and [13] respectively, we believe that they will also inherit the problems related to the implementation.

Finally, notice that in the setting of Dπ and Ambient explicit tagging of locations/data could seem useless, because no runtime check is ever performed. It might then appear more natural to leave the syntax untagged and record all type related information in the typing environment. The main drawback of this solution is that it would require a global (centralised) knowledge of all types. Tagging, instead, permits storing and inferring typing information locally, and keeps the formalisms closer to programmers' needs.

Pb **in** Node regions could be handled more dynamically by extending the calculi we presented with actions for adding and removing nodes from regions (in this way, e.g., nodes could choose whether to trust newly created ones). However, this more expressive framework would require additional runtime checks. In particular, none of the two guarantees illustrated at the beginning of Section 4.2.1 could then be issued after static checks; the type system would then only permit inferring the regions of the arguments of process actions, and render dynamic checks more efficient.

R **Wh** Much work has been recently devoted to designing languages for mobile processes that come equipped with security mechanisms based on, e.g., type systems [6, 13, 17, 28, 34] or control and data flow analysis [4, 22, 30, 45]. The approach presented here is related to both these techniques. It exploits a type system to ensure confinement of data, and guarantees that the semantics respects the annotations by relying on the typing phase. Typing keeps track of data movements with a technique similar to control flow analysis.

Our work has been inspired by that on Confined-λ [36], a higher-order functional language that supports distributed computing by allowing expressions at different localities to communicate via channels. To limit the movement of values, programmers can assign a type (i.e. a region) to them; a type system is defined that guarantees that each value can roam only within the allowed region. There are however some differences with our approach. First of all, we consider not only channels but also other communication media that require a more dynamic typing mechanism. Then, we permit annotating only the relevant data while in [36] a programmer must declare a type for any constant, function and channel. When typing a net, we do not rely on any form of global knowledge of the system; only the annotations in the process are considered. We can infer this information by the local use of channels/ambients (in Dπ and Ambients resp.) or by just examining the code (in CKLAIM). On the contrary, the type system in [36] assumes a global typing environment for handling shared channels; this somehow conflicts with the features of a global computing setting. Finally, we also give 'localised' formulations of the soundness theorem stating that well-typedness of a given subnet is preserved also in presence of untyped contexts. This is a crucial property for global computing systems where little assumptions on the behaviour of the context can be made.

Confinement has been also explored in the context of Java. In [49], confined types are introduced to confine classes and objects within specific packages. Then, in [50], a static type system based on confined types is defined for a Java-based calculus and its soundness is proved. Hence, in Java software modules play the role of our network regions, and confinement is associated rather with objects encapsulation than with movements of data and processes.

The group types, originally proposed for the Ambients calculus [13] and then recast to the π-calculus

[11], have purposes similar to our region annotations. A group type is just a name that can be dynamically created but cannot be communicated (i.e. the scope of a group name cannot be extended). It permits to control name visibility in different regions of a net: a fresh name belonging to a fresh group can never be communicated to any process outside the scope of the group. Group types can then be used to handle processes and ambients movements, and in general to prevent accidental or malicious leakage of private names without using more complex dependent types (see, e.g., [39]). Notice that restricted names can be handled in a more flexible way in our framework by readily adapting the use of groups or, even better, of the *b t m* from [39]. We leave these aspects for future work. However, differently from our approach, when exploiting groups or abstract names some global knowledge is still necessary for taking into account the types of the names occurring free in a net.

Group types have also been used in region-based memory management where the focus is on efficiency, rather than on distribution and mobility. For instance, in [18] a connection between memory regions and group types is established and a variant of the π-calculus equipped with group types is used as a device to simplify the proof of correctness of dynamic memory management.

Finally, we want to consider a lower level approach to protect visibility of data via *p* . Encrypted data can appear everywhere in the net, but can be effectively used only by those users that know the decryption key. At an abstract level, we can consider the content of an encrypted message to be visible only within the region containing the nodes knowing the decryption key. Thus, it might appear that our approach could be implemented by resorting to cryptographic primitives. However, we would like to stress an important difference. When encryption is used, the producer of encrypted data can control the access to (plain) data only by controlling visibility of the decryption key. But this can be hardly controlled: once a decryption key has been passed on, information leakage can reveal the key, thus breaking the controllability of data. By exploiting our approach, the data producer can decide in advance which are the users enabled to access the data; this information is preserved during any evolution of the system. However, it should be noticed that indirect information flows can be generated; for an account of these problems and some possible solutions we refer the interested reader to [31, 33].

5 Conclusions

In this deliverable, we have presented and discussed some typing approaches aimed at establishing security properties for open systems of global computers. The approaches we have presented rely on different languages and on different design choices underlying the type theory. Nevertheless, the results presented offer several possible alternatives in order to monitor the execution of a global computing application, ranging from hystory-based access control to dynamic typing and data secrecy. The work summarised here is an important contribution to the MIKADO project carried on by the people at Lisbon, Torino and Firenze.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of NDSSS'03*, pages 107–121, 2003.

[2] G. Boudol. Asynchrony and the p-calculus. Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[3] G. Boudol, I. Castellani, F. Germain, and M. Lacoste. Models of distribution and mobility: State of the art. Mikado Deliverable D1.1.1, 2002.

[4] C. Braghin, A. Cortesi, and R. Focardi. Security Boundaries in Mobile Ambients. *Computer Languages*, 28(1):101–127, Nov 2002.

[5] M. Bugliesi and G. Castagna. Behavioral typing for Safe Ambients. *Computer Languages*, 28(1):61 – 99, 2002.

[6] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[7] M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *Proceedings of CONCUR'04*, volume 3170 of *LNCS*, pages 225–239. Springer-Verlag, 2004.

[8] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication and Mobility Control in Boxed Ambients. To appear in *Information and Computation*. Extended and revised version of M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In FSTTCS'02, volume 2556 of LNCS, pages 71-84. Springer-Verlag, 2002., April 2003.

[9] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 51–94. Springer-Verlag, 1999.

[10] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer-Verlag, 1999.

[11] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In C. Palamidessi, editor, *Proc. of CONCUR 2000*, volume 1877 of *LNCS*, pages 365–379. Springer, 2000.

[12] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In J. van Leeuwen, O. Watanabe, M. Hagiya, and T. I. Peter D. Mosses, editors, *International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 333–347. Springer-Verlag, 2000.

[13] L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the ambient calculus. *Journal of Information and Computation*, 177(2):160–194, 2002.

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

[15] T. Chothia and I. Stark. A distributed pi-calculus with local areas of communication. In *ENTCS*, volume 41.

[16] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and R. Pugliese. Dynamic and local typing for mobile ambients. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2004. Available at `http://www.dsi.unifi.it/~pugliese /DOWNLOAD/dltma-full.pdf`.

[17] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In J. Harland, editor, *CATS 2003*, volume 78 of *ENTCS*. Elsevier, 2003.

[18] S. Dal-Zilio and A. D. Gordon. Region analysis and a pi-calculus with groups. In M. Nielsen and B. Rovan, editors, *Proceedings of MFCS 2000*, volume 1893 of *LNCS*, pages 409–424. Springer, 2000.

[19] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and mobility. *IEEE Trans. in Software Engineering*, 24(5):315–330, 1998.

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

[21] R. De Nicola, D. Gorla, and R. Pugliese. Confining data and processes in global computing applications. *Science of Computer Programming*. To appear.

[22] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Proc. of ASIAN'00*, volume 1961 of *LNCS*, pages 199–214. Springer, 2000.

[23] G. Edjlali, A. Anurag, and C. Vipin. History-based access-control for mobile code. In *Proceedings of CCS'98*.

[24] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, pages 87–95. ACM Press, 1999.

[25] G. Ferrari, E. Moggi, and R. Pugliese. Guardians for ambient-based monitoring. In V. Sassone, editor, *F-WAN*, volume 66 of *ENTCS*. Elsevier, 2002.

[26] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[27] E. Giovannetti. Type inference for mobile ambients in prolog. In *Proc. of CATS'04*, ENTCS. Elsevier, 2004.

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

[29] D. Gorla and R. Pugliese. Controlling data movement in global computing applications. In *Proc. of 19th Annual ACM-SIGAPP Symposium on Applied Computing (SAC'04)*. ACM Press, 2004.

[30] R. R. Hansen, J. G. Jensen, F. Nielson, and H. R. Nielson. Abstract interpretation of mobile ambients. In *Proc. of SAS 1999*, volume 1694 of *LNCS*, pages 134–148. Springer, 1999.

[31] M. Hennessy. The security pi-calculus and non-interference. In *Proc. of MFPS XIX*, ENTCS. Elsevier, 2003. Full version to appear in *Journal of Logic and Algebraic Programming*.

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

[33] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In U. Montanari, J. Rolim, and E. Welzl, editors, *Proc. of ICALP 2000*, volume 1853 of *LNCS*, pages 415–427. Springer, 2000.

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

[35] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.

[36] Z. D. Kirli. Confined mobile functions. In *Proc. of the 14th CSFW*, pages 283–294. IEEE Computer Society, 2001.

[37] F. Levi and D. Sangiorgi. Controlling interference in Ambients. *Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.

[38] C. Lhoussaine. Type inference for a distributed pi-calculus. In *Proc. of ESOP'03*, volume 2618 of *LNCS*, pages 253–268. Springer-Verlag, 2003. Full version to appear in *Science of Computer Programming*.

[39] C. Lhoussaine and V. Sassone. A dependently typed ambient calculus. In *Proceedings of ESOP'04*, volume 2986 of *LNCS*, pages 171–187. Springer-Verlag, 2004.

[40] F. Martins and A. Ravara. Typing migration control in *lsd*p. In Andrei Sabelfield, editor, *Proceedings of FCS'04*. TUCS, 2004.

[41] F. Martins and V. Vasconcelos. Controlling security policies in a distributed environment. DI/FCUL TR 04–01, 2004.

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

[43] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[44] G. Necula. Proof-Carrying Code. In *Proceedings of POPL '97*, pages 106–119. ACM Press, 1997.

[45] F. Nielson, H. R. Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In J. C. Baeten and S. Mauw, editors, *Proc. of CONCUR '99*, volume 1664 of *LNCS*, pages 463–477. Springer, 1999.

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

[47] A. Ravara, A. Matos, V. Vasconcelos, and L. Lopes. Lexically scoping distribution: what you see is what you get. In *FGC: Foundations of Global Computing*, volume 85(1) of *ENTCS*.

[48] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back*, volume 2000 of *LNCS*, pages 86–101. Springer, 2000.

[49] J. Vitek and B. Bokowski. Confined types in java. *Software - Practice and Experience*, 31(6):507–532, 2001.

[50] T. Zhao, J. Palsber, and J. Vitek. Lightweight confinement for featherweight java. In *Proc. of the 18th OOPSLA*, pages 135–148. ACM Press, 2003.

[51] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Proc. of FoSSaCS'00*, volume 1784 of *LNCS*, pages 375–390. Springer, 2000.

[52] E. Zwicky, S. Cooper, and D. Chapman. *Building Internet Firewalls, Second Edition*. O´Reilly & Associates, 2000.