Enforcing Security Policies via Types*

Daniele Gorla[1,2] Rosario Pugliese[2]

[1]Dipartimento Informatica, Università di Roma "La Sapienza"
[2]Dipartimento di Sistemi e Informatica, Università di Firenze

Abstract. Security is a key issue for distributed systems/applications with code mobility, like, e.g., e-commerce and on-line bank transactions. In a scenario with code mobility, traditional solutions based on cryptography cannot deal with all security issues and additional mechanisms are necessary. In this paper, we present a flexible and expressive type system for security for a calculus of distributed and mobile processes. The type system has been designed to supply real systems security features, like the assignment of different privileges to users over different data/resources. Type soundness is guaranteed by using a combination of static and dynamic checks, thus enforcing specific security policies on the use of resources. The usefulness of our approach is shown by modeling the simplified behaviour of a bank account management system.

Technical Report 05/2004
Dipartimento di Informatica
Università di Roma "La Sapienza"

* This work has been partially supported by EU within the FET - Global Computing initiative, project MIKADO IST-2001-32222, and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

1 Introduction

Code mobility is a fundamental aspect of global computing; however it gives rise to a lot of relevant security problems like, e.g., *secrecy* and *integrity* of data and program code. Indeed, in mobile distributed systems/applications, other than attacks to inter-process communication over the communication channels (e.g. traffic analysis, message modifications/forging), several other kinds of attacks could take place. For instance, malicious mobile processes can attempt to access private information, or modify private data of the nodes hosting them. Hence, a server receiving a mobile process for execution needs to impose strong requirements to ensure that the incoming process does not violate the secrecy and jeopardize the integrity of the information. Similarly, mobile processes need tools to ensure that their execution at the server node does not compromise their integrity (e.g. modification of process code) or secrecy (e.g. leak of sensible data). Such problems have increasingly importance due to the spreading of security critical applications, like, e.g., electronic commerce and on-line bank transactions. Moreover, global computing environments, like e.g. the Internet, are highly dynamic and open systems. In these environments static information could be partial, inaccurate or missing, therefore for ensuring security properties a certain amount of dynamic checks is needed (e.g. mobile agents should be dynamically checked at run-time when they migrate).

Code mobility strongly restricts a safe use of cryptography, that is one of the most used techniques for ensuring security in distributed systems. In fact, because of attacks like those mentioned before, we can hardly imagine to use mobile processes carrying confidential data (e.g. private keys) with them, or host nodes with classified information accessible to all incoming processes (whatever their source node be). Hence, the use of security mechanisms that back up and supplement cryptographic mechanisms becomes a major issue when developing systems of distributed and mobile processes where the compliance with some security policies must be guaranteed.

Several alternative approaches have been exploited to enforce security policies in distributed computing systems. The approaches may differ in the level of trust required, the flexibility of the enforced security policy and their costs to components producers and users. A comprehensive security framework could result from the combination of complementary features. Approaches like *code signing* and *sand-boxing* (for instance, consider the Java implementation of these concepts [22, 20]) have low costs but cannot enforce flexible security policies (signed components may behave in arbitrary ways and the user must trust the component producer, while sand-boxed components are isolated and cannot interact with each other).

Type systems can be sensible and flexible language-based security techniques, like [31] shows. Recently, a number of process/programming languages supporting process distribution and mobility have been designed that come equipped with type systems that guarantee some kind of security properties, see, e.g., [23, 15, 16, 24, 8]. However, to the best of our knowledge, the type system we present in this paper is the first that exploits the source of mobile processes for granting them different privileges over different kinds of data (thus, e.g., preventing dangerous operations over specific sensible data). These desirable features can be found in real systems like, e.g., UNIX, where different users can have different privileges and different files can be manipulated with different allowed operations.

Our type system permits expressing and enforcing security policies for controlling the access of host resources by possibly malicious mobile processes. It is expressly designed for the process calculus μKLAIM [21] that puts forward a programming paradigm where there is a clear separation between the programmer level and the net coordinator/administrator level. Programmers write processes, while coordinators write nets, hence manage the initial distribution of processes and set the security policies for accessing the resources. The policies are specified by assigning each node of a net a type expressing the operations a process is allowed to perform once spawned at it. Hence types are part of the language for configuring the underlying net architecture and must be taken into account in the language operational semantics. Other than to express security policies, types are used to record processes intended operations, but programmers are relieved from typing processes because this task is carried on by a static type inference system. By using a combination of static and dynamic type checking, our system guarantees the absence of run-time errors due to lack of privileges. As an application of our approach we model the simplified behaviour of a bank account management system where the compliance with the bank security policy must be enforced.

The rest of the paper is organized as follows. We present the syntax of μKLAIM in Section 2, its type system in Section 3, and its operational semantics in Section 4. Section 5 contains the main theoretical results of the paper, i.e. the *subject reduction* and *type safety* theorems. In Section 6, we illustrate an application of our approach to model a bank account management system. Finally, in Section 7 we point out a few concluding remarks and comment on related work.

2 The Process Language μKLAIM

In this section we briefly present the syntax and informally describe the semantics of μKLAIM [21], a calculus to program distributed and mobile processes communicating asynchronously via shared data.

The syntax of μKLAIM is reported in Table 1. We assume the existence of the following countable sets: $\mathcal{A}$, *process identifiers*, ranged over by $A, B, \ldots$; $\mathcal{L}$, *localities*, ranged over by l; $\mathcal{U}$, locality variables, ranged over by u; $\mathcal{V}$, *basic values*, ranged over by V. We let ℓ to range over $\mathcal{L} \cup \mathcal{U}$, x over value variables, π over sets of *capabilities* and Δ over *types* (capabilities and types are formally defined in Section 3).

The syntax of *expressions*, ranged over by e, is deliberately not specified; we just assume that expressions contain, at least, basic values and variables. *Localities* l are the addresses of nodes. *Tuples* t are sequences of actual fields f, that contain information items (expressions, localities or locality variables). Tuples are collected into multisets called *tuple spaces* (TSs, for short). *Templates* T are used to select tuples in a TS; they are sequences of actual and formal fields F. The latters are used to bind variables to values and are written $!\,x$ or $!\,u : \pi$ (the set of capabilities π constraints the use of the address dynamically bound to u and is crucial for the type checking).

Processes are built up from the inactive process **nil** and from the basic operations by using prefixing, parallel composition and process invocation. For the sake of simplicity, we assume that each process identifier A has a *single* defining equation $A \stackrel{\triangle}{=} P$ and all these equations are available at any locality of a net. Recursive behaviours can be modelled via process definitions.

$N ::= \mathbf{0}$	(empty net)	$a \ ::= \mathbf{read}(T)@\ell$	(process actions)
$\quad l ::^{\Delta} P$	(single node)	$\mathbf{in}(T)@\ell$	
$\quad N_1 \parallel N_2$	(net composition)	$\mathbf{out}(t)@\ell$	
		$\mathbf{eval}(P)@\ell$	
$P \ ::= \mathbf{nil}$	(null process)		
$\quad a.P$	(action prefixing)	$\mathbf{newloc}(u : \Delta)$	
$\quad P_1 \mid P_2$	(parallel composition)	$T \ ::= F \quad F, T$	(templates)
$\quad A$	(process invocation)	$F \ ::= f \quad !\,x \quad !\,u : \pi$	(template fields)
		$t \ ::= f \quad f, t$	(tuples)
$e \ ::= V \quad x \quad \ldots$	(expressions)		
		$f \ ::= e \quad \ell$	(tuple fields)

Table 1. μKlaim Syntax

μKlaim supplies five different basic operations, also called *actions*. $\mathbf{out}(t)@\ell$ adds the tuple *et* resulting from the evaluation[1] of t to the TS located at ℓ. Operation $\mathbf{eval}(Q)@\ell$ sends process Q for execution to ℓ, where a run-time typechecking of the incoming code will take place: if Q does not comply with ℓ's security policy the operation is blocked. Operation $\mathbf{in}(T)@\ell$ evaluates T and looks for a matching[2] tuple *et* in the TS located at ℓ; if *et* is found, it is withdrawn and the values it contains are used to replace the corresponding variables of T within the continuation process, otherwise the operation is suspended until a matching *et* is available. Operation **read** behaves similarly but leaves the accessed tuple *et* in the tuple space. Operation $\mathbf{newloc}(u : \Delta)$ dynamically creates a new net node with a fresh address whose security policy is specified by type Δ. The last operation is not indexed with an address because it always acts locally; all the other operations explicitly indicate the (possibly remote) address where they will take place.

Variables occurring in process terms can be *bound* by action prefixes. More precisely, prefixes $\mathbf{in}(T)@\ell.\_$ and $\mathbf{read}(T)@\ell.\_$ bind the variables in the formal fields of T, and prefix $\mathbf{newloc}(u : \delta).\_$ binds u. In process $a.P$, P is the scope of the bindings made by a. A variable that is not bound is called *free*. The sets $\textsc{bv}(P)$ and $\textsc{fv}(P)$ (of bound and free variables, resp., of P) are defined accordingly, and so is *α-conversion*, denoted $\equiv_\alpha$. In the sequel, we shall assume that bound variables in processes are all distinct and different from the free variables (by possibly applying alpha-conversion, this requirement can always be satisfied). Moreover, we shall consider only *closed* processes, i.e. processes without free variables.

Nets are finite collections of nodes where processes and tuple spaces can be allocated. A *node* is a triple $l ::^{\Delta} P$, where locality l is the address (i.e. network reference) of the node, P is the (parallel) process located at l and Δ is the type of the node, i.e. the specification of its access control policy. The nodes of a net can be thought of both as physically distributed machines and

[1] Tuple/template evaluation consists in replacing each expression with the value resulting from its evaluation.

[2] An evaluated tuple matches against an evaluated template if both have the same number of fields and corresponding fields match; formal fields of a given type match values of the same type and two values match only if identical.

(Alpha)	$\dfrac{N \equiv_\alpha N'}{N \equiv N'}$	(Id)	$N \parallel \mathbf{0} \equiv N$
(Com)	$N_1 \parallel N_2 \equiv N_2 \parallel N_1$	(Ass)	$N_1 \parallel (N_2 \parallel N_3) \equiv (N_1 \parallel N_2) \parallel N_3$
(Abs)	$l ::^{\Delta} P \equiv l ::^{\Delta} (P \vert \mathbf{nil})$	(Call)	$l ::^{\Delta} A \equiv l ::^{\Delta} P \quad$ if $A \triangleq P$

Table 2. Nets Structural Congruence

as logical partitions of the same machine. As we shall see in Section 4, the TS located at l is part of P because evaluated tuples are semantically represented as special processes.

We will identify nets which intuitively represent the same net. We therefore define *structural congruence* $\equiv$ to be the smallest congruence relation over nets that satisfies the laws in Table 2. Notice that $\equiv$ identifies only nets whose equality is immediately obvious from their syntactical structure and has nothing to do with the semantics of nets (which has still to be introduced and shall rely on structural congruence). If not differently specified, in the sequel we shall only consider *well-formed nets*, i.e. nets where pairwise distinct nodes have different addresses. This request is a simple way to guarantee that each network node has one security policy. However, our main results (see Section 5) do not rely on this hypothesis and still hold for generic nets.

3 A Capability-Based Type System

In this section we introduce a type system for μKlaim that permits granting different privileges to processes coming from different nodes and constraining the operations allowed over different kinds of data. Thus, for example, if l trusts l', then l security policy could accept processes coming from l' (that will be called l'-processes) and let them accessing any tuple in l's TS. If l' is not totally trusted, then l's security policy could grant l'-processes the capabilities for executing **in**/**read** only over tuples that do not contain classified data.

3.1 Capabilities

Capabilities are used to specify the allowed process operations and are formally defined as

$$\mathcal{C} \triangleq \{e, n\} \ \cup \ \{ \ \langle c, p \rangle \ : \ c \in \{i, r, o\} \ \wedge \ p \ \subseteq_{\text{fin}} \mathcal{P} \ \}$$

where $\mathcal{P} \triangleq (\mathcal{L} \cup \mathcal{V} \cup \{\mathbf{from}, -\})^+$ is the set of all *patterns*. Capabilities e and n enable process migration and node creation (i.e. operations **eval** and **newloc**, resp.). A capability of the form $\langle c, p \rangle$ enables the operation whose name's first character is c (i.e. **in** if c is i, and so on); operation arguments must comply with the finite set of patterns p if $p \neq \emptyset$, and are not restricted otherwise (in this case, we write c instead of $\langle c, \emptyset \rangle$). Like tuples and templates, *patterns* are finite, not empty sequences of fields; *pattern fields* may be localities, basic values, the reserved word **from** (denoting the last locality visited by a mobile process) and the 'don't care' symbol $-$ (denoting any template field). Thus, for instance, the capability $\langle \ i \ , \ \{(\text{“}\textit{public}\text{”}, -), (3, -, \mathbf{from})\} \ \rangle$ enables the operations $\mathbf{in}(\text{“}\textit{public}\text{”}, !x)@...$ and $\mathbf{in}(3, !u : \pi, l)@...$ for an l-process, while disables operation $\mathbf{in}(\text{“}\textit{private}\text{”}, !x)@...$.

$$\{\langle i,p\rangle\} \sqsubseteq_\Pi \{\langle r,p\rangle\} \qquad \frac{\mathcal{T}(p') \subseteq \mathcal{T}(p)}{\{\langle c,p\rangle\} \sqsubseteq_\Pi \{\langle c,p'\rangle\}} \qquad \frac{\pi_1 \subseteq \pi_2}{\pi_2 \sqsubseteq_\Pi \pi_1} \qquad \frac{\pi_1 \sqsubseteq_\Pi \pi_1' \quad \pi_2 \sqsubseteq_\Pi \pi_2'}{\pi_1 \cup \pi_2 \ \sqsubseteq_\Pi \ \pi_1' \cup \pi_2'}$$

Table 3. Capability Ordering Rules

We use π to denote a non-empty subset of $\mathcal{C}$ such that, if $\langle c,p\rangle \in \pi$ and $\langle c',p'\rangle \in \pi$, then $c \neq c'$. Π will denote the set of all these π.

We say that a template *complies with* a pattern if the template is obtained by replacing in the pattern all occurrences of **from** with a locality, and any occurrence of '$-$' with any template field allowed by the syntax. Given a non-empty set of patterns p, we write $\mathcal{T}(p)$ to denote the set of all templates complying with patterns in p. By definition, $\mathcal{T}(\emptyset)$ denotes the set of all templates. Since tuples are also templates (see Table 1), the previous definitions also apply to tuples.

Notice that the definition of pattern fields affects, via the relation 'complies with', the ability of our types to control the tuples accessed by process operations. However, our framework is largely independent of the choice of a specific set of fields. For instance, we could also permit fields of the form $-_\delta$, for any type δ of legal values, with the idea that, when defining the relation 'complies with', an occurrence of $-_\delta$ could be replaced by any value/variable of type δ. In μKLAIM, this corresponds to adding only fields $-_\mathcal{L}$ and $-_\mathcal{V}$; in this way, a finer control could be exercised on the tuples accessed by processes because we could distinguish between a tuple field containing a locality from one containing a basic value.

We now introduce an ordering between capabilities, $\sqsubseteq_\Pi$; formally, it is the least reflexive and transitive relation induced by the rules in Table 3. The chosen ordering relies on the following assumptions: (i) if a process is allowed to perform an **in** then it is also allowed to perform a **read** over the same arguments, (ii) if a process is allowed to perform a **read**/**in**/**out** over arguments complying with patterns in p then it is allowed to perform the same operation over arguments complying with any set of patterns p' that has at most the same 'complying templates' as p, and (iii) if a process owns a set of capabilities π_2 then it also owns any subset π_1.

3.2 Types

Types, ranged over by Δ, are functions of the form

$$\Delta : \mathcal{L} \cup \mathcal{U} \cup \{\mathbf{any}\} \rightarrow_{\text{fin}} \Big((\mathcal{L} \cup \mathcal{U} \cup \{\mathbf{any}, \mathbf{from}\} \ \rightarrow_{\text{fin}} \ \Pi \cup \emptyset) \cup \bot \Big)$$

where $\rightarrow_{\text{fin}}$ means that the function maps only a finite subset of its domain to significant values (i.e. values different from $\bot$ and $\emptyset$). With abuse of notation, we use $\bot$ to also denote the empty type, i.e. the function mapping all its domain to $\bot$. Moreover, by letting λ to range over $\mathcal{L} \cup \mathcal{U} \cup \{\mathbf{any}, \mathbf{from}\}$, we shall write a Δ different from $\bot$ as a non-empty list $[\lambda_i \mapsto [\lambda_{i,j} \mapsto \pi_{i,j}]_{j=1,\ldots,k_i}]_{i=1,\ldots,n}$. Types are used to express the security policies of nodes. Intuitively, if the type Δ of a node with address l contains the element $[l' \mapsto l'' \mapsto \pi]$, then l'-processes located at l are allowed to perform over l'' only the operations enabled by π. The reserved word **any** is used to refer any node of the net. If it occurs in the domain of Δ then it collects the privileges granted to processes coming from any node of the net (i.e. $[\mathbf{any} \mapsto l'' \mapsto \pi]$ grants all processes the privileges π over l''). If **any** is contained in the domain of $\Delta(l')$, for some l', then it is used for denoting the operations that l'-processes located at l are allowed to perform over any node of the net (i.e. $[l' \mapsto \mathbf{any} \mapsto \pi]$ grants

l'-processes the privileges π over all net nodes). The reserved word **from** stands for the last node visited by a process and is used to grant privileges over this node whatever it is; thus, for instance, $[\mathbf{any} \mapsto \mathbf{from} \mapsto \pi]$ grants l'-process spawned at l the privileges π over l'. The type $\bot$ expresses total absence of privileges.

We now introduce the notion of *sub-typing*. Since types are functions, the notion of subtyping is derived from the standard preorder over functions, by also using the pointwise union of functions, denoted by $\Cup$.

Definition 1. *The* subtyping *relation, $\leq$, is the least reflexive and transitive relation closed under the rule*

$$\frac{\forall\ \lambda \in dom(\Delta_1) : \Delta_1(\lambda) \preceq \Delta_2(\lambda) \Cup \Delta_2(\mathbf{any})}{\Delta_1 \leq \Delta_2}$$

where $\preceq$ is the least reflexive and transitive relation closed under the rule

$$\frac{\forall\ \lambda' \in dom(\Delta_1(\lambda)) : \Delta_2(\lambda)(\lambda') \cup \Delta_2(\lambda)(\mathbf{any}) \sqsubseteq_{\Pi} \Delta_1(\lambda)(\lambda')}{\Delta_1(\lambda) \preceq \Delta_2(\lambda)}$$

Thus, if $\Delta_1 \leq \Delta_2$, then Δ_1 is less permissive than Δ_2; moreover, $\bot \leq \Delta$ for any Δ, since $dom(\bot) = \emptyset$. We finally introduce the notion of *well-formed* types, that will be useful when proving soundness of our system.

Definition 2. *The type Δ is l–*well-formed *whenever the following conditions hold:*
1. *If* $\mathbf{from} \in dom(\Delta(\lambda))$ *then* $\lambda = \mathbf{any}$
2. *For each $\ell \in dom(\Delta)$, it holds that $\Delta(\ell) \preceq \Delta(l)$*
3. *For each $\lambda \in dom(\Delta(\mathbf{any}))$-$\{\mathbf{from}\}$, it holds that $\Delta(l)(\lambda) \sqsubseteq_{\Pi} \Delta(\mathbf{any})(\lambda)$*
4. $\Delta(l)(\mathbf{any}) \sqsubseteq_{\Pi} \Delta(\mathbf{any})(\mathbf{from})$.

Apart from technical reasons, this definition seems reasonable when considering Δ to be the type of locality l. Indeed, the first condition is not too restrictive, because the use of **from** is really necessary only when no knowledge of the last node visited by processes is available (i.e. when using **any**). The second condition says that l grants to ℓ-processes (for $\ell \in dom(\Delta)$) no more privileges than those granted to its local processes, i.e. those processes statically allocated at l. Finally, the last conditions are similar to the second one, but apply to processes coming from any node; in this case, it is also required that processes coming from any node own over the source node no more privileges than those owned by local processes over any node (condition 4.).

Remark 1. Notice that the syntax of types allows locality variables to occur within types. Basically, they are used when specifying the type of a node dynamically created for referring localities that will be dynamically determined. By exploiting this feature, we can write processes like the following: $\mathbf{in}(!u : ...)@... .\mathbf{newloc}(v : [u \mapsto v \mapsto \{r\}])$.

3.3 Static Type Checking

For each node of a net, say $l ::^{\Delta} P$, the static type checker analyzes the operations that P intends to perform when running at l and determines whether they are enabled by the access policy Δ or

$$\Gamma \vdash_{l} \mathbf{nil}$$

$$\frac{\Gamma(\ell) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle o, p \rangle \qquad t \in \mathcal{T}(p) \qquad \Gamma \vdash_{l} P}{\Gamma \vdash_{l} \mathbf{out}(t)@\ell.P}$$

$$\frac{\Gamma(\ell) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle i, p \rangle \qquad T \in \mathcal{T}(p) \qquad \mathit{upd}(\Gamma, T) \vdash_{l} P}{\Gamma \vdash_{l} \mathbf{in}(T)@\ell.P}$$

$$\frac{\Gamma(\ell) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle r, p \rangle \qquad T \in \mathcal{T}(p) \qquad \mathit{upd}(\Gamma, T) \vdash_{l} P}{\Gamma \vdash_{l} \mathbf{read}(T)@\ell.P}$$

$$\frac{\Gamma(\ell) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \{e\} \qquad \Gamma \vdash_{l} P}{\Gamma \vdash_{l} \mathbf{eval}(Q)@\ell.P}$$

$$\frac{\Gamma(l) \sqsubseteq_{\Pi} \{n\} \qquad \Delta \text{ is } u\text{–well-formed} \qquad \Gamma \uplus [u \mapsto (\Gamma(l) - \{n\})] \vdash_{l} P}{\Gamma \vdash_{l} \mathbf{newloc}(u : \Delta).P}$$

$$\frac{\Gamma \vdash_{l} P \qquad \Gamma \vdash_{l} Q}{\Gamma \vdash_{l} P \mid Q} \qquad\qquad \frac{\Gamma \vdash_{l} P}{\Gamma \vdash_{l} A} \text{ if } A \triangleq P$$

Table 4. Type Inference Rules

not (in fact, it is enough to consider $\Delta(l)$). To this aim, a *type context* Γ is a function of the form $\mathcal{L} \cup \mathcal{U} \cup \{\mathbf{any}\} \rightarrow_{\text{fin}} (\Pi \cup \emptyset)$. To update a type context with the type annotations specified within a template, we use the auxiliary function *upd* that behaves like the identity function for all fields but for formal fields binding locality variables. Formally, it is defined by:

$$\mathit{upd}(\Gamma, T) = \begin{cases} \mathit{upd}(\mathit{upd}(\Gamma, F), T') & \text{if } T = F, T' \\ \Gamma \uplus [u \mapsto \pi] & \text{if } T = !\, u : \pi, \\ \Gamma & \text{otherwise} \end{cases}$$

The type judgments for processes take the form $\Gamma \vdash_{l} P$, where the domain of Γ includes all the localities and all the free locality variables in P. The set of bindings for the localities in Γ implements the access policy of l for the processes statically located at l, while the remaining bindings record the type annotations for the locality variables that are free in P. Intuitively, the judgment $\Gamma \vdash_{l} P$ states that, within the context Γ, P can be safely executed once located at l.

Type judgments are inferred by using the rules in Table 4 that should be quite explicative. For operations **out**, **in**, **read** and **eval**, the inference requires the capability associated to the operation to be enabled by the capabilities owned over the target ℓ or over all the net sites. Instead, for operation **newloc**, the capability n must be owned by the site l executing the operation. Moreover, in this case, it is assumed that the creating node owns over the created one all the privileges it owns on itself (except, obviously, for the n capability).

We conclude this section by introducing the notion of *well-typed* net.

Definition 3. *A net N is* well-typed *if for each node $l ::^{\Delta} P$ in N it holds that Δ is l–well-formed and $\Delta(l) \vdash_{l} P$.*

$$match_{\mathit{\Delta}(l)}(V, V) \qquad match_{\mathit{\Delta}(l)}(l', l') \qquad match_{\mathit{\Delta}(l)}(!\,x, V)$$

$$\frac{\mathit{\Delta}(l)(l') \cup \mathit{\Delta}(l)(\mathbf{any}) \ \sqsubseteq_{\mathit{\Pi}} \ \pi}{match_{\mathit{\Delta}(l)}(!u : \pi, l')} \qquad \frac{match_{\mathit{\Delta}(l)}(eF, ef) \quad match_{\mathit{\Delta}(l)}(eT, et)}{match_{\mathit{\Delta}(l)}(\ (eF, eT)\ ,\ (ef, et)\ )}$$

Table 5. Matching Rules

4 Operational Semantics

We start by introducing a way to represent tuples and tuple spaces. Like in [15, 16], we model tuples as processes. To this aim, we extend the μKLAIM syntax with processes of the form $\mathbf{out}(et)$, where et stands for evaluated tuples. Tuples/templates evaluation function is written $\mathcal{E}[\![\,\cdot\,]\!]$ and simply replaces each expression occuring in $\cdot$ with its evaluation. Processes $\mathbf{out}(et)$ are similar to process **nil** because they do not perform any action and, thus, need no capability. Well-typedness of these processes is stated by the axiom

$$(\star) \qquad \mathit{\Gamma} \vdash_{l} \mathbf{out}(et)$$

that must be added to the rules in Table 4.

We then define the pattern-matching predicate *match*, used to select (evaluated) tuples from a tuple space according to (evaluated) templates. The predicate is defined by the rules in Table 5. It states that matching succeeds whenever the tuple and the template have the same number of fields and values in corresponding positions are identical. Moreover, it requires that for each formal field $!u : \pi$ the corresponding value l'' that will replace u is such that the security policy $\mathit{\Delta}$ of the node l where the **in**/**read** operation is performed allows local processes to perform all the operations enabled by π over l'', using if needed also the capabilities owned by l's static code over all the net.

Finally, the μKLAIM operational semantics is given by a net reduction relation, $\rightarrowtail$, specifying the basic computational steps and formalizing the informal behaviours sketched in Section 2. Because of the highly dynamic nature of our calculus, the operational semantics uses types to perform some dynamic checks; in these cases, the check occurs in the premises of an inference rule thus, if it fails, the rule cannot be used in the inference (i.e. the corresponding net reduction step is blocked).

$\rightarrowtail$ is the least relation induced by the rules in Table 6. Net reductions are defined over configurations of the form $L \vdash N$, where L is such that $\mathtt{loc}(N) \subseteq L \subset_{fin} \mathcal{L}$ and function $\mathtt{loc}(N)$, that could be easily defined by induction on the syntax of terms, returns the set of localities occurring in N. In a configuration $L \vdash N$, L keeps track of the localities in N and is needed to ensure global freshness of new addresses. For the sake of readability, when a reduction does not generate any fresh address we write $N \rightarrowtail N'$ instead of $L \vdash N \rightarrowtail L \vdash N'$.

Let us comment on the rules in Table 6. Rule (OUT) says that, before adding a tuple to a tuple space, the tuple must be evaluated. Rule (EVAL) says that a process is allowed to migrate only if it successfully passes a type checking. Indeed, the premise of the rule says that the migrating process Q must be checked against the union of the privileges that the security policy $\mathit{\Delta}'$ of the target node l' assignes to processes coming from l and to processes coming from any node (in this last case, occurrences of **from** must be interpreted as l, as stated by the syntactic substitution $\mathit{\Delta}'(\mathbf{any})[{}^{l}\!/\!_{\mathbf{from}}]$ of **from** with l in function $\mathit{\Delta}'(\mathbf{any})$). Rules (IN) and (READ) say that the process performing the

$$\text{(Out)}\quad \frac{et = \mathcal{E}[\![\, t \,]\!]}{l ::^{\Delta} \mathbf{out}(t)@l'.P \parallel l' ::^{\delta'} P' \rightarrowtail l ::^{\Delta} P \parallel l' ::^{\Delta'} P' | \mathbf{out}(et)}$$

$$\text{(Eval)}\quad \frac{\Delta'(l) \Cup (\Delta'(\mathbf{any})[{}^{l}\!/\mathbf{from}]) \vdash_{l'} Q}{l ::^{\Delta} \mathbf{eval}(Q)@l'.P \parallel l' ::^{\Delta'} P' \rightarrowtail l ::^{\Delta} P \parallel l' ::^{\Delta'} P' | Q}$$

$$\text{(In)}\quad \frac{match_{\Delta(l)}(\mathcal{E}[\![\, T \,]\!], et)}{l ::^{\Delta} \mathbf{in}(T)@l'.P \parallel l' ::^{\Delta'} \mathbf{out}(et) \ \rightarrowtail \ l ::^{\Delta} P[{}^{et}\!/T] \parallel l' ::^{\Delta'} \mathbf{nil}}$$

$$\text{(Read)}\quad \frac{match_{\Delta(l)}(\mathcal{E}[\![\, T \,]\!], et)}{l ::^{\Delta} \mathbf{read}(T)@l'.P \parallel l' ::^{\Delta'} \mathbf{out}(et) \ \rightarrowtail \ l ::^{\Delta} P[{}^{et}\!/T] \parallel l' ::^{\Delta'} \mathbf{out}(et)}$$

$$\text{(New)}\quad \frac{l' \notin L \qquad \Delta'[{}^{l'}\!/u] \leq (\Delta^{-n} \Cup \ [l \mapsto l' \mapsto \Delta(l)(l)])}{L \vdash l ::^{\Delta} \mathbf{newloc}(u : \Delta').P \rightarrowtail L \cup \{l'\} \vdash l ::^{\Delta \,\Cup\, [l \mapsto l' \mapsto (\Delta(l)(l) - \{n\})]} P[{}^{l'}\!/u] \parallel l' ::^{\Delta'[l'/u]} \mathbf{nil}}$$

$$\text{(Split)}\quad \frac{L \vdash l ::^{\Delta} P \ \parallel l ::^{\Delta} Q \ \parallel \ N \rightarrowtail L' \vdash l ::^{\Delta'} P' \ \parallel l ::^{\Delta} Q' \ \parallel \ N'}{L \vdash l ::^{\Delta} P|Q \ \parallel \ N \rightarrowtail L' \vdash l ::^{\Delta'} P'|Q' \ \parallel \ N'}$$

$$\text{(Par)}\quad \frac{L \vdash N_1 \rightarrowtail L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \rightarrowtail L' \vdash N_1' \parallel N_2}$$

$$\text{(Struct)}\quad \frac{N \equiv N_1 \qquad L \vdash N_1 \rightarrowtail L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \rightarrowtail L' \vdash N'}$$

Table 6. μKLAIM Operational Semantics

operation can proceed only if pattern-matching succeeds. In this case, all the formal fields of T are replaced with the corresponding values of et in the continuation process P (written $P[{}^{et}\!/T]$). Notice that, since types occurring in the continuation process may contain locality variables, substitutions must also be applied to such types. In rule (New) the set L of localities already in use is exploited to choose a fresh address l' for naming the new node. Notice that, once created, the address of the new node is not known to any other node in the net. Thus, it can be used by the creating process as a sort of *private* resource. In order to enable the creation, the specified access policy Δ', after modification with substitution $[{}^{l'}\!/u]$, must be in agreement with the access policy Δ of the node executing the operation (Δ^{-n} denotes the access policy Δ except that $\Delta^{-n}(\cdot)(l)$ is defined to be $\Delta(\cdot)(l) - \{n\}$) extended with the ability of performing over l' all the operations allowed locally (a part for **newloc**, of course). This is needed to prevent a malicious node l from forging capabilities by creating a new node with powerful privileges where sending a malicious process that takes advantage of capabilities not owned by l. Rule (Split) is used to split the parallel processes running at a node thus enabling the application of the rules previously mentioned that, in fact, can only be used when there is only one process running at l. Technically, a parallel over processes is transformed into a parallel over nodes. There is a subtlety with rule (Split): in fact, the nets in the premise of the rule are not well-formed according to the definition of Section 2 because they contain nodes with the same address (locality l)[3]. However, the transition in the conclusion preserves well-formedness. Rules (Par) and (Struct) are standard. The former says that, if part of a composed net evolves,

[3] This feature permits a compact and general formulation of the reduction rules without the need of explicitly considering all the parallel processes running at a node and of distinguishing between local and remote operations.

the whole net evolves accordingly. The latter says that structural congruent nets have the same reductions.

To conclude, we state two properties of the operational semantics. The first one says that nets well-formedness is preserved along reductions, the second one states a relationship between the set L and the net N in a configuration $L \vdash N$ (the proofs are omitted since they are very similar to those in [21]).

Proposition 1. *If N is well-formed and $\mathtt{loc}(N) \vdash N \rightarrowtail L' \vdash N'$ then N' is well-formed.*

Proposition 2. *If $\mathtt{loc}(N) \vdash N \rightarrowtail L' \vdash N'$ then $\mathtt{loc}(N') \subseteq L'$.*

5 Type Soundness

We can now state two standard results for type systems, namely, *subject reduction* and *type safety*. The former means that well-typedness is an invariant of the operational semantics; the latter means that well-typed nets are free from immediate run-time errors. In our framework, such errors would arise when processes attempt to execute operations that are disabled by the security policy of the node where they are running. The two properties together amount to saying that well-typed nets never give rise to run-time errors due to misuse of access privileges.

To prove subject reduction, we first give three preliminary results used in its proof. The first two state that the static inference is not affected by a uniform renaming of free locality variables with localities and by using more permissive type contexts; the last one states that well-typedness is preserved by structural congruence.

Lemma 1 (Substitutivity). *If $\Gamma \vdash_{\overline{l}} P$ then $\Gamma\sigma \vdash_{\overline{l}} P\sigma$, for each substitution $\sigma : \mathcal{U} \rightarrow_{\text{fin}} \mathcal{L}$ such that $dom(\sigma) \cap \text{BV}(P) = \emptyset$.*

Proof: The proof is by induction on length of the inference of the type judgement. The base cases (i.e., the first rule of Table 4 and rule $(\star)$) are obvious. Let us examine the case in which the last rule used deals with action prefixing for some action a different from **newloc** (the **newloc**, parallel composition and process invocation cases are easier).

$a = \mathbf{out}(t)@\ell$**.** By hypothesis, $\Gamma \vdash_{\overline{l}} Q$, $\Gamma(\ell) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle o, p \rangle$ and $t \in \mathcal{T}(p)$. By induction, we have that $\Gamma\sigma \vdash_{\overline{l}} Q\sigma$; moreover, $\Gamma\sigma(\ell\sigma) \cup \Gamma\sigma(\mathbf{any}) \sqsubseteq_{\Pi} \langle o, p \rangle$ (since substitutions do not affect patterns) and $t\sigma \in \mathcal{T}(p)$ (straightforward by definition of compliance with a pattern). By static inference, this sufficies to conclude the desired $\Gamma\sigma \vdash_{\overline{l}} (a.Q)\sigma$.

$a = \mathbf{eval}(Q)@\ell$**.** Similar.

$a = \mathbf{in}(T)@\ell$ or $a = \mathbf{read}(T)@\ell$. The only difference with the **out** case is that the hypothesis is $upd(\Gamma, T) \vdash_{\overline{l}} Q$. By induction, we have that $(upd(\Gamma, T))\sigma \vdash_{\overline{l}} Q\sigma$; it is easy to prove that $(upd(\Gamma, T))\sigma = upd(\Gamma\sigma, T\sigma)$ and to conclude. □

Lemma 2 (Weakening). *If $\Gamma \vdash_{\overline{l}} P$ then $\Gamma' \vdash_{\overline{l}} P$, for each Γ' such that $\Gamma \preceq \Gamma'$.*

Proof: By induction on the length of the inference for $\Gamma \vdash_{\overline{l}} P$, we prove that each step of such inference is still legal, once we replace Γ with Γ'. The base case is trivial. Let us only consider the

in inductive case (the others are similar or simpler). By static inference, we have that there exists a pattern p such that $\Gamma(\ell) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle i, p\rangle$ and $T \in \mathcal{T}(p)$. By hypothesis, $\Gamma'(\ell) \sqsubseteq_{\Pi} \Gamma(\ell)$ and $\Gamma'(\mathbf{any}) \sqsubseteq_{\Pi} \Gamma(\mathbf{any})$; by transitivity, $\Gamma'(\ell) \cup \Gamma'(\mathbf{any}) \sqsubseteq_{\Pi} \langle i, p\rangle$. We are still left with proving that $upd(\Gamma, T) \vdash_{l} P$ implies $upd(\Gamma', T) \vdash_{l} P$; but this easily follows by induction and by the fact that $upd(\Gamma, T) \preceq upd(\Gamma', T)$ (that is true because of definition of function upd). □

Lemma 3. *If N is well-typed and $N \equiv N'$ then N' is well-typed.*

Proof: By inspection of the axioms in Table 2.

(Alpha). The thesis directly follows from the fact that the static type inference is not affected if we uniformly rename bound variables within a net.

(Id), (Com), (Ass) and (Call). Obvious.

(Abs). By hypothesis, $\Delta(l) \vdash_{l} P$. Moreover, we have that $\Delta'(l') \vdash_{l'} \mathbf{nil}$ for each Δ' and l'. Hence, it follows that $\Delta \vdash_{l} P|\mathbf{nil}$. □

Theorem 1 (Subject Reduction). *If N is well-typed and $\mathtt{loc}(N) \vdash N \succ\!\!\longrightarrow L' \vdash N'$ then N' is well-typed.*

Proof: The proof proceeds by induction on the length of the inference of $\mathtt{loc}(N) \vdash N \succ\!\!\longrightarrow L' \vdash N'$. Notice that the sets of localities $\mathtt{loc}(N)$ and L' do not play any role in what follows (namely, they do not affect the definition of well-typed net) and will be ignored in the rest of the proof.

Base Step : By case analysis on the axioms (i.e. the first five rules) of Table 6.

(Out). By hypothesis, we have that $\Delta(l) \vdash_{l} \mathbf{out}(t)@l'.P$. Due to the form of the process involved, the second rule of Table 4 has been the last one applied to deduce the type judgement; hence we also have that $\Delta(l) \vdash_{l} P$. Moreover, $\Delta' \vdash_{l'} P' \mid \mathbf{out}(et)$ by applying the rule for paralel composition of Table 4 to $\Delta'(l') \vdash_{l'} P'$, that holds by hypothesis, and to the axiom $\Delta'(l') \vdash_{l'} \mathbf{out}(et)$. This sufficies to conclude well-typedness of N'.

(Eval). Well-typedness of node $l ::^{\Delta} P$ is inferred like in the previous case. We are only left to prove that $\Delta'(l') \vdash_{l'} Q$ since, by hypothesis, $\Delta'(l') \vdash_{l'} P'$. By the premise of rule (Eval), we have that $\Delta'(l) \Cup (\Delta'(\mathbf{any})[^{l}/_{\mathbf{from}}]) \vdash_{l'} Q$. If we prove that $\Delta'(l) \Cup (\Delta'(\mathbf{any})[^{l}/_{\mathbf{from}}]) \preceq \Delta'(l')$, we can conclude by using Lemma 2. Indeed, it sufficies to prove that both $\Delta'(l)$ and $\Delta'(\mathbf{any})[^{l}/_{\mathbf{from}}]$ are subtypes of $\Delta'(l')$, since trivially the pointwise extension of partial functions respects this property. By l'–well-formedness of type Δ', we have that:

- $\Delta'(l) \preceq \Delta'(l')$ (by condition 2. of Definition 2)
- for each $\lambda \in dom(\Delta'(\mathbf{any})[^{l}/_{\mathbf{from}}])$, using conditions 3. and 4. of Definition 2, it holds that:
  - if $\lambda \neq l$ then $(\Delta'(\mathbf{any})[^{l}/_{\mathbf{from}}])(\lambda) = \Delta'(\mathbf{any})(\lambda) \; {}_{\Pi}\!\sqsupseteq \; \Delta'(l')(\lambda)$
  - otherwise, $(\Delta'(\mathbf{any})[^{l}/_{\mathbf{from}}])(l) = \Delta'(\mathbf{any})(l) \cup \Delta'(\mathbf{any})(\mathbf{from}) \; {}_{\Pi}\!\sqsupseteq \; \Delta'(l')(l) \cup \Delta'(l')(\mathbf{any})$

  These facts amount to say that $\Delta'(\mathbf{any})[^{l}/_{\mathbf{from}}] \preceq \Delta'(l')$.

(In). To prove that $\Delta(l) \vdash_{l} P[^{et}/_{T}]$, we use the fact that $upd(\Delta(l), T) \vdash_{l} P$ (that holds by hypothesis). By definition, if $\{u_i\}_{i \in I}$ are the locality variables bound by T and π_i are the privileges associated to u_i in T, we have that $upd(\Delta(l), T) = \Delta(l) \Cup [u_i \mapsto \pi_i]_{i \in I}$. Moreover, by the premise of rule (In), we have that $match_{\Delta(l)}(\mathcal{E}[\![\, T \,]\!], et)$ and hence $\Delta(l)(l_i) \cup \Delta(l)(\mathbf{any}) \sqsubseteq_{\Pi} \pi_i$ for each

$$(\textsc{ErrAct})\ \frac{\Delta(l)(loc(a)) \cup \Delta(l)(\mathbf{any})\ \not\sqsubseteq_{\Pi} \{cap(a)\}}{l ::^{\Delta} a.P \uparrow l}$$

$$(\textsc{ErrPar})\ \frac{N \uparrow l}{N \parallel N' \uparrow l} \qquad (\textsc{ErrStr})\ \frac{N \equiv N' \quad N' \uparrow l}{N \uparrow l}$$

Table 7. Run-Time Error

$i \in I$. This fact means that $[l_i \mapsto \pi_i]_{i \in I} \preceq \Delta(l)$. Now, let $\sigma \triangleq [{}^{l_i}\!/\!_{u_i}]_{i \in I}$; hence $P[{}^{et}\!/\!_{T}] = P\sigma$. Putting together all these results, we have that $upd(\Delta(l), T) \vdash_{l} P$ implies $(upd(\Delta(l), T))\sigma = \Delta(l) \uplus [l_i \mapsto \pi_i]_{i \in I} \vdash_{l} P\sigma$ (by Lemma 1 and by the fact that $dom(\Delta(l)) \cap \{l_i\}_{i \in I} = \emptyset$), and hence $\Delta(l) \vdash_{l} P[{}^{et}\!/\!_{T}]$ (by Lemma 2 and definition of σ).

(Read). Similar to the previous case.

(New). First of all, we have to prove that the types occurring in the resulting net are still well-formed. Type $\Delta \uplus [l \mapsto l' \mapsto (\Delta(l)(l) - \{n\})]$ is l–well-formed, since it is obtained from the l–well-formed type Δ by adding privileges to l-processes; type $\Delta'[{}^{l'}\!/\!_{u}]$ is l'–well-formed since, by static inference, we have that Δ' is u–well-formed.

Then, we are left to prove that $\Delta \uplus [l \mapsto l' \mapsto (\Delta(l)(l) - \{n\})] \vdash_{l} P[{}^{l'}\!/\!_{u}]$. By hypothesis and definition of the static inference for the **newloc** prefix, it holds that $\Delta(l) \uplus [u \mapsto (\Delta(l)(l) - \{n\})] \vdash_{l} P$. By using Lemma 1, we can conclude the desired $(\Delta(l) \uplus [u \mapsto (\Delta(l)(l) - \{n\})])[{}^{l'}\!/\!_{u}] = \Delta \uplus [l \mapsto l' \mapsto (\Delta(l)(l) - \{n\})] \vdash_{l} P[{}^{l'}\!/\!_{u}]$.

Inductive Step : By case analysis on the last applied operational rule of Table 6.

(Split). By hypothesis, we have that $\Delta(l) \vdash_{l} P|Q$; hence, we also have that $\Delta(l) \vdash_{l} P$ and that $\Delta(l) \vdash_{l} Q$. Thus, the net $l ::^{\Delta} P \parallel l ::^{\Delta} Q \parallel N$ is well-typed. By induction, we get that $l ::^{\Delta'} P' \parallel l ::^{\Delta} Q' \parallel N'$ is well-typed; finally, using Lemma 2 applied to $\Delta(l) \preceq \Delta'(l)$, we get that $\Delta'(l) \vdash_{l} Q'$ and hence we obtain the thesis.

(Par). By hypothesis, $N_1 \parallel N_2$ is well-typed, hence N_1 and N_2 are well-typed too. Now, by induction, N_1' is well-typed and hence $N_1' \parallel N_2$ is well-typed.

(Struct). From the hypothesis, N is well-typed and $N \equiv N_1$; by Lemma 3, it follows that N_1 is well-typed too. Now, by induction, we get that N_2 is well-typed. From this fact and from the hypothesis $N_2 \equiv N'$, again by Lemma 3, it follows that N' is well-typed. □

To prove type safety, we firstly introduce *run-time errors*; they are defined by the rules in Table 7 in terms of predicate $N \uparrow l$ that holds true when, within the net N, a process P located at a node with address l attempts to perform an action that is not allowed by the access policy of the node. There, we use functions $loc(a)$ and $cap(a)$ to denote, resp., the target locality of action a and the least capability needed to safely perform the action; moreover, $\not\sqsubseteq_{\Pi}$ stands for the negation of predicate $\sqsubseteq_{\Pi}$. The rules are straightforward.

Theorem 2 (Type Safety). *If N is well-typed then $N \uparrow l$ for no $l \in \mathtt{loc}(N)$.*

Proof: We proceed by contradiction and prove, by induction on the length of the proof of $N \uparrow l$, that if $N \uparrow l$ for some $l \in \mathtt{loc}(N)$ then N is not well-typed.

Base Step : The only axiom of Table 7 is rule (ErrAct). Let us now distinguish two cases:

- $a.P$ was part of a l'-process migrated in l. In this case, it could not be $\Delta(l') \Cup (\Delta(\mathbf{any})[{}^{l'}\!/\mathbf{from}]) \vdash_{\overline{l}} a.P$, otherwise, by Lemma 2, $\Delta(l) \vdash_{\overline{l}} a.P$ and in particular $\Delta(l)(loc(a)) \cup \Delta(l)(\mathbf{any}) \sqsubseteq_{\Pi} \{cap(a)\}$. But in this case, the premise of rule (Eval) in Table 6 is not satisfied and hence the migration of the l'-process containing $a.P$ was blocked. This is in contradiction with assumed origin of process $a.P$.
- $a.P$ was part of a static process of l. In this case, it must be that $\Delta(l)(loc(a)) \cup \Delta(l)(\mathbf{any}) \not\sqsubseteq_{\Pi} \{cap(a)\}$, thus falsifying the premise of the static inference rule for action a. This is in contradiction with the well-typedness hypothesis.

Inductive Step : By case analysis on the last error rule used.

(ErrPar). From the premise $N \uparrow l$ of the rule, by induction, we have that N is not well-typed. Hence, by definition, $N \parallel N'$ is not well-typed too.

(ErrStr). From the premise $N' \uparrow l$ of the rule, by induction, we have that N' is not well-typed. Then the thesis follows from the premise $N \equiv N'$ by using Lemma 3. □

Therefore, executable nets cannot immediately give rise to run-time errors. Now, by combining together the results shown so far, we get that executable nets never generate run-time errors along sequences of reductions (usually, we denote with $\succ\!\!\longrightarrow^{\star}$ the reflexive and transitive closure of $\succ\!\!\longrightarrow$).

Corollary 1 (Global Type Soundness). *If N is well-typed and $\mathtt{loc}(N) \vdash N \succ\!\!\longrightarrow^{\star} L' \vdash N'$ then $N \uparrow l$ for no $l \in \mathtt{loc}(N)$.*

Proof: The proof proceeds by induction on the length of $\mathtt{loc}(N) \vdash N \succ\!\!\longrightarrow^{\star} L' \vdash N'$. The base step is trivial, the inductive step follows from Theorems 1 and 2, by using Proposition 2. □

Type soundness is one of the main goal of a type system. However, in our framework it is formulated in terms of a property requiring the typing of whole nets. While this could be acceptable for LANs, where the number of hosts usually is relatively small, it is unreasonable for WANs. When dealing with larger nets, it is certainly more realistic to reason in terms of parts of the whole net. Hence, we put forward a more *local* formulation of our properties and results. To this aim, we define the *restriction* of a net N to a set of localities D, written N_D, as the subnet obtained from N by deleting all nodes whose addresses are not in D. The wanted local type soundness result can be formulated as follows.

Theorem 3 (Local Type Soundness). *Let N be a net and $D \subseteq \mathtt{loc}(N)$. If N_D is well-typed and $\mathtt{loc}(N) \vdash N \succ\!\!\longrightarrow^{\star} L' \vdash N'$ then for no $l \in D$ it holds that $N' \uparrow l$.*

The proof is similar to the proof of Corollary 1. In fact, the local type soundness is enforced by the dynamic checking performed when processes migrate, which prevents to move ill-typed processes into N_D.

6 A Bank Account Management System

In this section, we use our approach to model the simplified behaviour of a bank account management system. For ensuring compliance with the security policy of the bank some aspects of our setting, such as the possibility of granting different privileges to processes coming from different source nodes and the dynamic type checking of mobile processes when they migrate, have proved to be crucial.

We suppose that a bank is located at a node with address l_B and can receive and manage requests coming from many users located at nodes with addresses l_U, $l_{U'}$, ... The bank must provide the users with typical account managing operations: opening/closing accounts, putting/getting money in/from accounts, and making statements of accounts. For simplicity, we shall omit some details and technical operations that in reality take place, like, e.g., the charge of taxes, dealing with improper operations like the attempt of getting more money than that really available, ...

For the sake of readability, in the rest of this section we will omit trailing occurrences of process **nil**, and use parameterized process definitions (that can be easily implemented in our setting using **out**/**in** operations to pass/recover the parameters), integer values (to denote, e.g., amounts of money) and strings (to identify the various operations).

For permitting the bank to check the operations that users intend to perform, we assume that users cannot perform remote operations over l_B except for sending processes. Hence, if a user U wants to require an operation to the bank, it has to send a process to l_B (thus virtually moving to the bank) which will interact locally with the proper operation handler. The user process, once it has been accepted (i.e. after its compliance with the bank security policy has been checked), can require the operation by locally producing a tuple whose first field contains the name of the operation and whose second field contains the address of the user node (used to identify the user that made the request). Depending on the operation, the tuple could have other fields containing the amount of money involved in the operation and the account receiving the money.

The node implementing the bank is illustrated in Table 8. First, the bank creates a new node that will contain its clients accounts, stored as tuples of the form $(userAddress, amount)$. This node acts just as a repository for tuples and will not be used for spawning processes, thus it has assigned the empty type $\perp$. Then, five different handler processes, one for each kind of operation, are concurrently spawned. Each handler continuously waits for a request. When such a request arrives, the proper handler executes its task by remotely accessing the reserved locality and then reports locally a confirmation of action completion. The client process performing the request waits for such a confirmation and then brings it back to its original locality. This last operation is performed by means of a migration thus providing the user node with the chance of controlling the operation.

Notice that, by taking advantage of the semantics of μKLAIM operations, the simple handlers of Table 8 implement the mutual exclusion needed to ensure the correctness of concurrent operations over shared data. Indeed, once a handler H has withdrawn the tuple representing an account (i.e. once H has locked the account), in order to proceed in their tasks, all the other handlers have to wait for H to write the updated tuple (i.e. for H to release the lock).

The security policy Δ_B is so defined that 'sensible' operations over the accounts of a user U (like getting some money and reading/closing the account) can only be requested by l_U-processes, while operations like putting some money can be requested by processes coming from any node. Moreover, the only remote operation processes are allowed to perform is to came back to their source site. Therefore, a l_U-process can request to the bank sensible operations only over U's accounts and can deliver the confirmations only to l_U. Typical processes acting on behalf of a user U are illustrated

$$l_B ::^{\Delta_B} \mathbf{newloc}(u:\bot).\big(\mathit{OpenH}(u) \mid \mathit{PutH}(u) \mid \mathit{GetH}(u) \mid \mathit{ReadH}(u) \mid \mathit{CloseH}(u)\big)$$

where:

$$\begin{array}{rcl}
\mathit{OpenH}(u) & \triangleq & \mathbf{in}(\text{“}\mathit{open}\text{”}, !x, !y)@l_B. \\
 & & \quad (\mathit{OpenH}(u) \mid \mathbf{out}(x,y)@u.\mathbf{out}(\text{“}\mathit{OKopen}\text{”}, x, y)@l_B) \\
\mathit{PutH}(u) & \triangleq & \mathbf{in}(\text{“}\mathit{put}\text{”}, !x, !y, !w)@l_B. \\
 & & \quad (\mathit{PutH}(u) \mid \mathbf{in}(w, !z)@u.\mathbf{out}(w, z+y)@u.\mathbf{out}(\text{“}\mathit{OKput}\text{”}, x, y, w)@l_B) \\
\mathit{GetH}(u) & \triangleq & \mathbf{in}(\text{“}\mathit{get}\text{”}, !x, !y)@l_B. \\
 & & \quad (\mathit{GetH}(u) \mid \mathbf{in}(x, !z)@u.\mathbf{out}(x, z-y)@u.\mathbf{out}(\text{“}\mathit{OKget}\text{”}, x, y)@l_B) \\
\mathit{ReadH}(u) & \triangleq & \mathbf{in}(\text{“}\mathit{read}\text{”}, !x)@l_B. \\
 & & \quad (\mathit{ReadH}(u) \mid \mathbf{read}(x, !y)@u.\mathbf{out}(\text{“}\mathit{OKread}\text{”}, x, y)@l_B) \\
\mathit{CloseH}(u) & \triangleq & \mathbf{in}(\text{“}\mathit{close}\text{”}, !x)@l_B. \\
 & & \quad (\mathit{CloseH}(u) \mid \mathbf{in}(x, !y)@u.\mathbf{out}(\text{“}\mathit{OKclose}\text{”}, x, y)@l_B)
\end{array}$$

$$\begin{array}{rl}
\Delta_B \triangleq & [\,l_B \mapsto [\,l_B \mapsto \{i, o, r, n\}, \\
 & \qquad\qquad \mathbf{any} \mapsto \{e\}\ ]\ , \\
 & \ \ \mathbf{any} \mapsto [\,\mathbf{from} \mapsto \{e\}, \\
 & \qquad\qquad l_B \mapsto \{\ \langle\ o\ , \{\ (\text{“}\mathit{open}\text{”}, \mathbf{from}, -), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{put}\text{”}, \mathbf{from}, -, -), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{get}\text{”}, \mathbf{from}, -), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{read}\text{”}, \mathbf{from}), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{close}\text{”}, \mathbf{from}) \\
 & \qquad\qquad\qquad\qquad\ \ \}\ \rangle, \\
 & \qquad\qquad\qquad \langle\ i\ , \{\ (\text{“}\mathit{OKopen}\text{”}, \mathbf{from}, -), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{OKput}\text{”}, \mathbf{from}, -, -), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{OKget}\text{”}, \mathbf{from}, -), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{OKread}\text{”}, \mathbf{from}, -), \\
 & \qquad\qquad\qquad\qquad\quad (\text{“}\mathit{OKclose}\text{”}, \mathbf{from}, -) \\
 & \qquad\qquad\qquad\qquad\ \ \}\ \rangle \\
 & \qquad\quad] \\
 & \ ]
\end{array}$$

Table 8. The node implementing the bank

in Table 9, where the parameter s denotes an amount of money and the parameter $l_{U'}$ denotes an account.

The only possibility for a malicious node to illegally access U's accounts is to pass through l_U, using a process like $\mathbf{eval}(\mathbf{eval}(\mathit{MaliciousReq})@l_B)@l_U$. Hence, U has to protect itself from these attacks by granting an e capability over l_B only to processes coming from totally trusted nodes: the security policy of l_U must contain the element $[l \mapsto l_B \mapsto \{e\}]$ only if U trusts the user located at l. However, U can trust l only if U trusts all l' trusted by l (in fact, a node trusted by l can send to l a process that is then allowed to spawn a process at U containing requests on U's accounts).

Finally, notice that only the handler processes can access the node dynamically created whose address, say l_S, is bound to u. Indeed, when such node is created, the operational semantics dy-

$$
\begin{aligned}
\mathit{OpenR}(s) &\triangleq \mathbf{eval}(\mathbf{out}(\text{“}\mathit{open}\text{”}, l_U, s)@l_B.\mathbf{in}(\text{“}\mathit{OKopen}\text{”}, l_U, s)@l_B.\\
&\qquad \mathbf{eval}(\mathbf{out}(\text{“}\mathit{OKopen}\text{”}, s)@l_U)@l_U)@l_B\\
\mathit{PutR}(s, l_{U'}) &\triangleq \mathbf{eval}(\mathbf{out}(\text{“}\mathit{put}\text{”}, l_U, s, l_{U'})@l_B.\mathbf{in}(\text{“}\mathit{OKput}\text{”}, l_U, s, l_{U'})@l_B.\\
&\qquad \mathbf{eval}(\mathbf{out}(\text{“}\mathit{OKput}\text{”}, s, l_{U'})@l_U)@l_U)@l_B\\
\mathit{GetR}(s) &\triangleq \mathbf{eval}(\mathbf{out}(\text{“}\mathit{get}\text{”}, l_U, s)@l_B.\mathbf{in}(\text{“}\mathit{OKget}\text{”}, l_U, s)@l_B.\\
&\qquad \mathbf{eval}(\mathbf{out}(\text{“}\mathit{OKget}\text{”}, s)@l_U)@l_U)@l_B\\
\mathit{ReadR} &\triangleq \mathbf{eval}(\mathbf{out}(\text{“}\mathit{read}\text{”}, l_U)@l_B.\mathbf{in}(\text{“}\mathit{OKread}\text{”}, l_U, !x)@l_B.\\
&\qquad \mathbf{eval}(\mathbf{out}(\text{“}\mathit{OKread}\text{”}, x)@l_U)@l_U)@l_B\\
\mathit{CloseR} &\triangleq \mathbf{eval}(\mathbf{out}(\text{“}\mathit{close}\text{”}, l_U)@l_B.\mathbf{in}(\text{“}\mathit{OKclose}\text{”}, l_U, !x)@l_B.\\
&\qquad \mathbf{eval}(\mathbf{out}(\text{“}\mathit{OKclose}\text{”}, x)@l_U)@l_U)@l_B
\end{aligned}
$$

Table 9. Processes of a user U requesting bank operations

namically extends Δ_B with $[l_B \mapsto l_S \mapsto \{i, r, o\}]$ thus enabling all the processes initially allocated at l_B to perform **in**/**read**/**out** operations over l_S.

7 Concluding Remarks

We presented a new capability based type system for the calculus μKLAIM [21] which controls data/resource access and process mobility in a flexible and expressive way. It has been designed to supply real systems security features, e.g. granting different privileges to processes coming from different nodes and constraining the operations allowed over different kinds of data/resources. Due to the highly dynamic nature of distributed and mobile systems/applications, our framework uses a combination of static and dynamic type checking to guarantee compliance with net security policies. As a future work we plan to integrate in μKLAIM other security mechanisms, like e.g. those based on cryptographic techniques, both for the establishment of secure channels, and for process code security and authentication.

The choice of the process calculus μKLAIM [21], that is at the core of the programming language KLAIM [14] and hence is based on the Linda [19, 11] coordination model, is motivated by the fact that μKLAIM has a number of features that make it appealing also for network computing environments where, in general, connections are not stable and host machines are heterogenous. Indeed, it permits *time uncoupling* (tuples life time is independent of the producer process life time), *destination uncoupling* (the producer of a tuple does not need to know the future use or the destination of that tuple) and *space uncoupling* (communicating processes need to know a single interface, i.e. the operations over the tuple space). As shown in [17], where several messaging models for mobile processes are examined, the *blackboard* approach, of which tuple space based models are variants, is one of the most appreciated, also because of its flexibility. Evidence of the success gained by the tuple space paradigm is given by the many tuple space based run-time systems, both from industries, e.g. JavaSpaces [32, 2] and TSpaces [35], and from universities, e.g. PageSpace [13], WCL [30], Lime [28] and TuCSoN [27].

Many type systems for guaranteeing security properties have been proposed for process calculi with distribution and mobility, but, as far as we know, ours is the first one implementing such fine grained policies. Among those type systems more strictly related to security, we mention those disciplining the types of the values exchanged in communications [9, 3, 24], those for controlling

Ambients [10] mobility and ability to be opened [6, 7, 26, 18, 12], that for controlling resource access via policies for mandatory access control [4], that for checking that all processes that intend to perform inputs at a given channel are co-located [36], that for controlling the effect of transmitted process abstractions over local channels [37], and that for restricting the mobility of values/processes only to some part of a distributed system [25].

The research line closest to ours is that on the Dπ-calculus [24], a distributed version of the π-calculus equipped with a type system to control access rights of mobile processes over located resources (i.e. communication channels). Like μKLAIM, the Dπ-calculus relies on a flat net architecture; however, differently from μKLAIM, communication is local and channel-based, types describe permissions to use channels, and the net architecture is not independent from the processes involved. [23, 29] present two improved type systems for the Dπ-calculus that permit establishing well-typedness of part of a net. This is similar to our local type soundness result that, however, has been obtained by using only local type information.

[36] presents D$\pi\lambda$, a process calculus that results from the integration of the call-by-value λ-calculus and the π-calculus, together with primitives for process distribution and remote process creation. Apart from the higher order and channel-based communication, the main difference with μKLAIM is that D$\pi\lambda$ localities are anonymous (i.e. not explicitly referrable by processes) and simply used to express process distribution. In [37], a fine-grained type system for D$\pi\lambda$ is defined that permits controlling the effect of transmitted process abstractions (parameterized with respect to channel names) over local channels. Processes are assigned fine-grained types that, like interfaces, record the channels to which processes have access together with the corresponding capabilities, and process abstractions are assigned dependent functional types that abstract from channel names and types. This use of types is similar to that of μKLAIM.

Finally, a number of process calculi base their security policies on transmission of encrypted data over communication channels so that only those processes knowing the proper keys can access these information. [1, 33, 5] present this approach in various settings, but none of them consider process distribution and mobility.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
2. K. Arnold, E. Freeman, and S. Hupfer. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.
3. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS 2001*, number 2215 in LNCS, pages 38–63. Springer, 2001.
4. M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *Concur 2001*, number 2154 in LNCS, pages 102–120. Springer, 2001.
5. N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro. Secspaces: a data-driven coordination model for environments open to untrusted agents. In *To appear in FOCLASA'02*, ENTCS, 2002.
6. L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In J. Wiederman, P. van Emde Boas, and M. Nielsen, editors, *Proceedings of ICALP '99*, volume 1644 of *LNCS*, pages 230–239. Springer, 1999.

7. L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Proceedings of TCS 2000*, volume 1872 of *LNCS*, pages 333–347. IFIP, Springer, 2000.
8. L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the ambient calculus. *Journal of Information and Computation*, 2002.
9. L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings of POPL '99*, pages 79–92.
10. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
11. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
12. G. Castagna, G. Ghelli, and F. Z. Nardelli. Typing mobility in the seal calculus. In *Concur 2001*, number 2154 in LNCS, pages 82–101. Springer, 2001.
13. P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–366, 1998.
14. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
15. R. De Nicola, G. Ferrari, and R. Pugliese. Types as Specifications of Access Policies. In Vitek and Jensen [34], pages 117–146.
16. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
17. D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proc. of ISADS 2001*, pages 278–286. IEEE, 2001.
18. M. Dezani-Ciancaglini and I. Salvo. Security types for mobile safe ambients. In *ASIAN Computing Sciece Conference - ASIAN'00*, volume 1961 of *LNCS*, pages 215–236. Springer, 2000.
19. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
20. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, Reading, MA, USA, 1999.
21. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2002. Available at `http://rap.dsi.unifi.it/~pugliese/DOWNLOAD/muklaim-full.pdf`.
22. G. M. Graw and E. Felten. *Securing Java.* John Wiley and Son, 1999.
23. M. Hennessy and J. Riely. Type-Safe Execution of Mobile Agents in Anonymous Networks. In Vitek and Jensen [34], pages 95–115.
24. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.
25. D. Kirli. Confined mobile functions. In *Proc. of the 14th IEEE Computer Security Foundations Workshop.* IEEE Computer Society, 2001.
26. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL '00*, pages 352–364. ACM, Jan. 2000.
27. A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):251–269, 1999. Special Issue on Coordination Mechanisms and Patterns for Web Agents.
28. G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, 1999.
29. J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL '99*, pages 93–104.
30. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.

31. F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back. Conference on the Occasion of Dagstuhl's 10th Anniversary*, LNCS 2000, pages 86–101. Springer, 2000.
32. Sun Microsystems. Javaspace specification. available at: `http://java.sun.com/`, 1999.
33. J. Vitek, C. Bryce, and M. Oriol. Coordinationg processes with secure spaces. *Science of Computer Programming*, 2002.
34. J. Vitek and C. Jensen, editors. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer-Verlag, 1999.
35. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998.
36. N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *CONCUR '99*, LNCS 1664, pages 557–572. Springer-Verlag, 1999.
37. N. Yoshida and M. Hennessy. Assigning types to processes. In *Proceedings of LICS'00*, pages 334–348. IEEE, Computer Society Press, June 2000.