Resource Access and Mobility Control with Dynamic Privileges Acquisition[⋆]

Daniele Gorla and Rosario Pugliese

Dipartimento di Sistemi e Informatica, Università di Firenze
{gorla,pugliese}@dsi.unifi.it

Abstract. μKLAIM is a process language that permits programming distributed systems made up of several mobile components interacting through multiple distributed tuple spaces. We present the language and a type system for controlling the activities, e.g. access to resources and mobility, of the processes in a net. By dealing with privileges acquisition, the type system enables dynamic variations of security policies. We exploit a combination of static and dynamic type checking, and of in-lined reference monitoring, to guarantee absence of run-time errors due to lack of privileges and state two type soundness results: one involves whole nets, the other is relative to subnets of larger nets.

1 Introduction

Process mobility is a fundamental aspect of global computing; however it gives rise to a lot of relevant security problems. Recently, a number of languages for mobile processes have been designed that come equipped with security mechanisms (at compilation and/or at run-time) based on, e.g., type systems, control and data flow analysis and proof carrying code.

Our starting point is KLAIM [9], an experimental language specifically designed to program distributed systems made up of several mobile components interacting through multiple distributed tuple spaces, and its capability-based type system [10] for controlling access to resources and mobility of processes. KLAIM has been implemented [2] by exploiting Java and has proved to be suitable for programming a wide range of distributed applications with agents and code mobility. In KLAIM, the network infrastructure is clearly distinguishable from user processes and explicitly modelled, which we believe gives a proper description of the computer systems we are interested to. KLAIM communication mechanism rests on an extension of the basic Linda coordination model [13] with multiple distributed tuple spaces. General 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. SUN JavaSpaces [1] and IBM T Spaces [22], and from universities, e.g. PageSpace [8], WCL [21], Lime [19] and TuCSoN [18].

⋆ Work partially supported by EU 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.

Klaim programming paradigm enjoys a number of properties, such as time uncoupling, destination uncoupling, space uncoupling, modularity, scalability and flexibility, that make the language appealing for open distributed systems and network computing environments (see, e.g., [11,14]), where, in general, connections are not stable and host machines are heterogenous. In conclusion, we think it is worthwhile to investigate the Klaim paradigm, also because its peculiar aspects about interprocess communication and network modelling distinguish it from the most popular and studied process languages.

The major contribution of this paper is the introduction of a calculus, called μKlaim, with process distribution and mobility, and of a relative type system for controlling process activities. μKlaim is at the core of Klaim and has a simpler syntax (without higher-order communication, with only one kind of addresses, without allocation environments, and without parameterized process definitions) and operational semantics. Moreover, it has a cleaner and powerful type system (types only record local information), that enables dynamic modifications of security policies and process privileges, and run-time type checking of programs, or part of them. In fact, static verification is useful in many circumstances since it avoids the use of dynamic mechanisms, thus improving system performances. However, it is hardly sufficient in highly dynamic systems, like e.g. open systems and the Internet, where it could restrict privileges and capabilities more than needed, thus unnecessarily reducing the expressive power (and the capabilities) of mobile processes. To deal with open systems, a certain amount of dynamic checking is needed (e.g. mobile processes should be dynamically checked at run-time when they migrate), also for taking into account that in these environments typing information could be partial, inaccurate or missing. Furthermore, extensive dynamic checking along with mechanisms supporting modifications at run-time of security polices and process privileges turn out to be essential for dealing with pervasive network applications, like e.g. those for e-commerce.

The μKlaim type system allows processes to be first partially verified and then executed in a more efficient and flexible way, rather than to run inefficiently because of massive run-time checks. Each network node has its own security policy that affects the evolution of the overall system and, thus, must be taken into account when defining the operational semantics. Types are used to express security policies in terms of capabilities (there is one capability for each process operation), hence they are part of the language for configuring the underlying net architecture. Moreover, 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.

Because of lack of space, we shall omit from this extended abstract several details and all proofs, and present a version of the calculus where communications exchange tuples with only one field; a thorough presentation can be found in [14].

Table 1. μKlaim Syntax

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

2 μKlaim Syntax

The syntax of μKlaim, given in Table 1, is parameterized with respect to the following syntactic sets, which we assume to be countable and pairwise disjoint: $\mathcal{A}$, of *process identifiers*, ranged over by $A, B, \ldots$; $\mathcal{L}$, of *localities*, ranged over by l; $\mathcal{U}$, of *locality variables*, ranged over by u. We use ℓ to range over $\mathcal{L} \cup \mathcal{U}$, V over basic values, x over value variables, π over sets of *capabilities*, δ over *types*, and μ over *capability specifications*.

The exact syntax of *expressions*, e, is deliberately not specified; we just assume that expressions contain, at least, basic values and variables. *Localities*, l, are the addresses (i.e. network references) of nodes. *Tuples*, t, contain expressions, localities or locality variables. In particular, $\ell : \mu$ points out a capability specification μ that permits dynamically determining the set of capabilities granted along with address ℓ. *Templates*, T, are used to select tuples. In particular, parameters $!\,x$ or $!\,u : \pi$ (the set of capabilities π constraints the use of the address dynamically bound to u) are used to bind variables to values.

Processes are the μKlaim active computational units and can perform a few basic operations over tuple spaces and nodes: retrieve/place (evaluated) tuples from/into a tuple space, send processes for execution on (possibly remote) nodes, and create new nodes. Processes are built up from the stuck process **nil** and from the basic operations by using action prefixing, parallel composition and process definition. It is assumed that each process identifier A has a *single* defining equation $A \triangleq P$. Of course, process defining equations should migrate along with invoking processes; however, for the sake of simplicity, we do not explicitly model migration of defining equations (that could be implemented like class code migration in [2]) and assume that they are available at any locality of a net.

Variables occurring in process terms can be *bound* by action prefixes **in**/**read**/**newloc**. For example, $\mathbf{in}(!\,u : \pi)@\ell.\_$ and $\mathbf{newloc}(u : \delta).\_$ bind u, while $\mathbf{in}(!\,x)@\ell.\_$ binds x. In process $a.P$, P is the scope of the binding made by a; we call *free* the variables in P that are not bound and accordingly define *α-conversion*. In the sequel, we shall assume that bound variables in processes are all distinct and different from the free variables (by possibly applying α-

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 quadruple $l ::_{\Sigma}^{\delta} P$, where locality l is the address of the node, P is the (parallel) process located at l, Σ is the set of process defining equations $\{A_1 \triangleq P_1, \ldots, A_n \triangleq P_n\}$ (with $A_i \neq A_j$ if $i \neq j$) that are valid at l, and δ is the type of the node, i.e. the specification of its access control policy. The *tuple space* (*TS*) located at l is part of P because, as we will see in Section 4, evaluated tuples are represented as special processes. In the sequel, we shall omit Σ whenever it plays no role.

We will identify nets which intuitively represent the same net. We therefore define *structural congruence* $\equiv$ to be the smallest congruence relation over nets equating α-convertible nets, stating that '$\|$' is commutative and associative and that **nil** is the identity for '$|$'. If not differently specified, in the sequel we shall only consider *well-formed nets*, i.e. nets where pairwise distinct nodes have different addresses.

Capabilities are elements of set $\{r, i, o, e, n\}$, where each symbol corresponds to the operation whose name begins with it; e.g. r denotes the capability of executing a **read** operation. We use Π, ranged over by π, to denote the set formed by the subsets of $\{r, i, o, e, n\}$.

Types, ranged over by δ, are functions of the form $\delta : \mathcal{L} \cup \mathcal{U} \rightarrow_{\text{fin}} \Pi$, where $\rightarrow_{\text{fin}}$ means that the function maps only a finite subset of its domain to non-empty sets. Notation $[\ell_i \mapsto \pi_i]_{\ell_i \in D}$ stands for the type δ such that $\delta(\ell)$ is π_i if $\ell = \ell_i \in D$ and is $\emptyset$ otherwise. The *extension* of δ_1 with δ_2, written $\delta_1[\delta_2]$, is the type δ' such that $\delta'(\ell) = \delta_1(\ell) \cup \delta_2(\ell)$ for each $\ell \in \mathcal{L} \cup \mathcal{U}$.

Capability specifications, ranged over by μ, are partial functions with finite non-empty domain of the form $\mu : \mathcal{L} \cup \mathcal{U} \rightharpoonup \Pi \cup \overline{\Pi}$, where $\overline{\Pi} \triangleq \{\overline{\pi} : \pi \in \Pi\}$. For capability specifications, we adopt a notation similar to that used for types, but now $[\ell_i \mapsto p_i]_{\ell_i \in D}$ (where $p_i \in \Pi \cup \overline{\Pi}$) stands for the capability specification μ such that $dom(\mu) = D$ and $\mu(\ell_i) = p_i$. Capability specifications are used, mainly in **out** operations, to identify sets of capabilities depending on the type at run-time of the node where processes run. In fact, when a process P running, say, at l wants to output a location l' along with some privileges, it is important to guarantee that P cannot grant larger privileges over l' than those owned by l. Since, in general, the latters can be determined only at run-time (because they depend on the privileges acquired by l over l' during the computation), capability specifications provide a way to statically express this fact.

3 A Capability-Based Type System

We start introducing a subtyping relation, $\preceq$. It relies on an ordering over sets of capabilities stating that, if $\pi_1 \sqsubseteq_{\Pi} \pi_2$, then π_1 enables at least the actions enabled by π_2. The type theory we develop is parametric with respect to the used capability ordering; here, for the sake of simplicity, we let $\sqsubseteq_{\Pi}$ to be the reverse subset inclusion. Now, we define $\preceq$ by letting $\delta_1 \preceq \delta_2$ whenever $\delta_2(\ell) \sqsubseteq_{\Pi} \delta_1(\ell)$

for each $\ell \in \mathcal{L} \cup \mathcal{U}$ (which is the standard preorder over functions). Thus, if $\delta_1 \preceq \delta_2$, then δ_1 is less permissive than δ_2.

Let us now present the static inference system. Informally, for each node, say $l ::_{\Sigma}^{\delta} P$, of a net, the inference system checks that all process identifiers occurring in P are defined in Σ and determines whether the actions that P intends to perform when running at l are enabled by the access policy δ or not. For example, capability e can be used to control process mobility: P can migrate to l' only if $[l' \mapsto \{e\}]$ is a subtype of δ. However, because l can dynamically acquire privileges when P performs **in**/**read** actions, some actions that can be permissible at run-time could be statically illegal. For this reason, if P intends to perform an action not allowed by δ, the static inference system cannot reject the process since the capability necessary to perform the action could in principle be dynamically acquired by l. In such cases, the inference system simply *marks* the action to require its dynamic checking. The marking mechanism never applies to actions whose targets are locality variables bound by **in**/**read**, because such actions can be statically checked, thus alleviating the burden of dynamic checking and improving system performance. In fact, according to the syntax, whenever a locality variable u is bound by an action **in**/**read**, u is annotated with a set of capabilities π that specifies the operations that the continuation process is allowed to perform by using u as the target address.

We therefore extend the μKlaim syntax to include marked actions, where a marked action is a normal μKlaim action which is underlined to require a dynamic checking of the corresponding capability. Formally, we extend the syntactic category of processes as $P ::= \ldots \mid \underline{a}.P$. We will write $\underline{P}$ ($\underline{N}$, resp.) to emphasize that process P (net N, resp.) may contain marked actions.

A *type context* Γ is a type. 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 templates but for those binding locality variables. In this case, we have $upd(\Gamma, \,!\,u : \pi) = \Gamma[u \mapsto \pi]$. Hence, if T is a tuple, then $upd(\Gamma, T) = \Gamma$. To have more compact inference rules for judgments, we found it convenient to extend function *upd* to encompass the case that the second argument is a process and let $upd(\Gamma, P) = \Gamma$.

Type judgments for processes take the form $\Gamma \vdash_{l}^{\Sigma} P \rhd \underline{P}$. In Γ, the bindings from localities to non-empty sets implement the access policy of the node with address l, while the bindings from locality variables to non-empty sets record the type annotations for the variables that are free (i.e. have been freed) in P. Intuitively, the judgment $\Gamma \vdash_{l}^{\Sigma} P \rhd \underline{P}$ states that, within the context Γ, when P is located at l, the unmarked actions in $\underline{P}$ are admissible w.r.t. Γ and all process identifiers occurring in P are defined in Σ.

Type judgments are inferred by using the rules in Table 2. Given an action a, we use $arg(a)$ to denote its argument, $tgt(a)$ its target location and $cap(a)$ the capability corresponding to a. Moreover, we mark actions by using function

$$mark_{\Gamma}(a) = \begin{cases} a & \text{if } \Gamma(tgt(a)) \sqsubseteq_{\Pi} \{cap(a)\} \\ \underline{a} & \text{if } \Gamma(tgt(a)) \not\sqsubseteq_{\Pi} \{cap(a)\} \text{ and } tgt(a) \in \mathcal{L} \end{cases}$$

Table 2. μKlaim Type Inference Rules

$$(1)\quad \Gamma \vdash^{\Sigma}_{l} \mathbf{nil} \triangleright \mathbf{nil} \qquad\qquad (2)\quad \frac{\Gamma \vdash^{\Sigma}_{l} P \triangleright \underline{P} \qquad \Gamma \vdash^{\Sigma}_{l} Q \triangleright \underline{Q}}{\Gamma \vdash^{\Sigma}_{l} P \mid Q \triangleright \underline{P} \mid \underline{Q}}$$

$$(3)\quad \frac{\Sigma = \Sigma' \cup \{A \triangleq P\}}{\Gamma \vdash^{\Sigma}_{l} A \triangleright A} \qquad\qquad (4)\quad \frac{cap(a) \neq n \qquad upd(\Gamma, arg(a)) \vdash^{\Sigma}_{l} P \triangleright \underline{P}}{\Gamma \vdash^{\Sigma}_{l} a.P \triangleright mark_{\Gamma}(a).\underline{P}}$$

$$(5)\quad \frac{\Gamma(l) \sqsubseteq_{\Pi} \{n\} \qquad \Gamma[u \mapsto (\Gamma(l) - \{n\})] \vdash^{\Sigma}_{l} P \triangleright \underline{P}}{\Gamma \vdash^{\Sigma}_{l} \mathbf{newloc}(u : \delta).P \triangleright \mathbf{newloc}(u : \delta).\underline{P}}$$

where $\not\sqsubseteq_{\Pi}$ denotes the negation of $\sqsubseteq_{\Pi}$. Condition $tgt(a) \in \mathcal{L}$ distinguishes actions using localities as target from those using variables, marking the formers and rejecting the latters (as previously explained). The rules in Table 2 should be quite explicative, we only remark a few points. Rule (3) says that a process identifier always successfully passes the static type checking provided that it is defined in Σ. Rule (4) deals with action prefixing. Notice that, in case of action **eval**, the argument process is not statically checked because the locality where the process will be sent for execution, and hence the access policy against which the process has to be checked, cannot be, in general, statically known. Action **newloc** is dealt with differently from the other actions by rule (5) and is always statically checked (i.e. it is never marked). Indeed, **newloc** is always performed locally and the corresponding capability cannot be dynamically acquired. Finally, notice that the creating node owns over the created one all the privileges it owns on itself (except, obviously, for the n capability).

Definition 1. *A net is* well–typed *if for each node $l ::^{\delta}_{\Sigma} P$, with $\Sigma = \{A_1 \triangleq P_1, \ldots, A_n \triangleq P_n\}$, there exist $P', P'_1, \ldots, P'_n$ such that $\delta \vdash^{\Sigma}_{l} P \triangleright P'$ and $\delta \vdash^{\Sigma}_{l} P_i \triangleright P'_i$, for each $i \in \{1, ..., n\}$.*

4 μKlaim Operational Semantics

An important ingredient we need for defining the operational semantics is a way to represent evaluated tuples and TSs. Like in [9], we model tuples as processes. To this aim, we extend the μKlaim syntax with processes of the form $\langle et \rangle$ (et stands for *evaluated tuple*), that similarly to process **nil** perform no action (and, thus, need no capability). Well-typedness of these auxiliary processes is stated by the axiom

$$(\star) \qquad \Gamma \vdash^{\Sigma}_{l} \langle et \rangle \triangleright \langle et \rangle$$

that is added to the rules in Table 2.

Only evaluated tuples can be added to a TS and, similarly, templates must be evaluated before being used to retrieve tuples. Hence we define the *tuple/template evaluation* function $\mathcal{T}[\![\, \cdot \,]\!]_{\delta}$ as the identity, except for

$$\mathcal{T}[\![\, e \,]\!]_{\delta} \;=\; \mathcal{E}[\![\, e \,]\!] \qquad\qquad \mathcal{T}[\![\, l : \mu \,]\!]_{\delta} \;=\; l : [\![\, \mu \,]\!]_{\delta(l)-\{n\}}$$

Table 3. Capability Specifications Evaluation Function

$$\begin{aligned}
[\![\ [l' \mapsto \pi']\ ]\!]_\pi &= [l' \mapsto \pi \cap \pi'] \\
[\![\ [l' \mapsto \overline{\pi}']\ ]\!]_\pi &= [l' \mapsto (\pi - \pi')] \\
[\![\ \mu_1[\mu_2]\ ]\!]_\pi &= (\ [\![\ \mu_1\ ]\!]_\pi\ )[\ [\![\ \mu_2\ ]\!]_\pi\ ]
\end{aligned}$$

where function $\mathcal{E}[\![\cdot]\!]$ evaluates expressions (thus it depends on the kind of allowed expressions and, hence, is left unspecified). $\mathcal{T}[\![\cdot]\!]_\delta$ takes as a parameter the type (i.e. access policy specification) of the node where the evaluation will take place and accordingly evaluates the contained capability specifications by using function $[\![\cdot]\!]_\pi$ (defined by the rules in Table 3). The latter is parameterized with respect to the set of capabilities owned by the node where the evaluation takes place over the locality which the capability specification being interpreted is associated to. Notice that, since actions **newloc** are always performed locally, the corresponding capability n is never transmitted. For this reason, the parameter of the interpretation function for capability specification does never contain n. The first rule ensures that no more privileges over a given l' than those owned by l are passed, while the second rule replaces $\overline{\pi}'$ with the complement of π' with respect to π, the set of capabilities used as a parameter of the evaluation function.

The *matching* function $match_l^\delta$, used to select evaluated tuples from a TS according to evaluated templates, is defined by the rules in Table 4. Function $match_l^\delta$ is parameterized with the locality l and the security policy δ of the node where it is invoked. A successful matching returns a type, used to extend the type of the node executing the matching with the capabilities granted by the (producer of the) tuple, and a substitution, used to assign values to variables in the (continuation of the) process invoking the matching. The first two rules say that two values match only if identical and that a value parameter match any value. Rule (3) requires that, for a matching to take place, the locality of the node where the **read**/**in** is executing must occur in the type specification associated to the locality being accessed. Rule (4) ensures that if a **read**/**in** executing at l looks for a locality where to perform the actions enabled by π, then, for selecting locality l', it must hold that the union of the privileges over l' owned by l and of the privileges over l' granted to l by the tuple enables the actions enabled by π. The privileges granted by the tuple are then used to enrich the capabilities of l over l'. Notice that (4) succeeds only if $l \in dom(\mu)$; this requirement, like that in the premise of rule (3), permits controlling immediate access to tuples (see Section 6).

Finally, the μKlaim operational semantics is given by a net reduction relation, $\rightarrowtail$, which is the least relation induced by the rules in Table 5. Net reductions are defined over configurations of the form $L \vdash N$, 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 $L \vdash N$, L keeps track of the localities in N and is needed to ensure global freshness of new addresses and, thus, to guarantee that well-formedness is preserved along reductions. For the sake of readability, when

Table 4. Matching Rules

(1) $match_l^\delta(V,V) = \langle [\,], \epsilon \rangle$	(2) $match_l^\delta(!\,x,V) = \langle [\,], [V/x] \rangle$
(3) $\dfrac{l \in dom(\mu_2)}{match_l^\delta(l' : \mu_1, l' : \mu_2) = \langle [\,], \epsilon \rangle}$	(4) $\dfrac{\delta(l') \cup \mu(l) \sqsubseteq_\Pi \pi}{match_l^\delta(!\,u : \pi, l' : \mu) = \langle [l' \mapsto \pi], [l'/u] \rangle}$

a reduction does not generate any fresh addresses we write $N \rightarrowtail N'$ instead of $L \vdash N \rightarrowtail L \vdash N'$; moreover, we also omit the sets of process defining equations from the nodes in N when they are irrelevant.

Let us now comment on the most significant rules in Table 5. Rule (Eval) says that a process is allowed to migrate only if it successfully passes a type checking against the access policy of the target node. During this preliminary check, some process actions could be marked to be effectively checked when being executed. Rules (In) and (Read) say that the process performing the operation can proceed only if matching succeeds. In this case, the access policy of the receiving node is enriched with the type returned by the matching mechanism and the substitution returned along with the type is applied to the continuation (and the type annotations therein) of the process performing the operation. 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 defined as follows: $\delta^{-n}(l) = \delta(l) - \{n\}$ and $\delta^{-n}(l'') = \delta(l'')$ for every $l'' \neq l$) extended with the ability to perform 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. Hereafter, we write $\overline{\Sigma}$ to denote the set Σ of process defining equations where all marks have been removed. Thus, notation $\delta'[l'/u] \vdash_{l'}^{\Sigma} \overline{\Sigma} \rhd \Sigma'$ means that the set of process defining equations is checkable under the access policy of the new node and returns Σ'. Rule (Mark) says that the in-lined security monitor stops execution whenever the privilege for performing a is missing. 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.

5 Type Soundness

We start introducing the notion of *executable nets* that, intuitively, are nets already containing all necessary marks (as if they have already passed a static type checking phase). The second clause of the definition accounts for the assumption that all process defining equations are available everywhere (but, in general, are differently marked because checked against different access policies).

Table 5. μKlaim Operational Semantics

(Out)	$\dfrac{et = \mathcal{T}[\![\, t \,]\!]_\delta}{l ::^{\delta} \mathbf{out}(t)@l'.P \parallel l' ::^{\delta'} P' \rightarrowtail l ::^{\delta} P \parallel l' ::^{\delta'} P' \mid \langle et \rangle}$
(Eval)	$\dfrac{\delta' \vdash^{\Sigma'}_{l'} Q \rhd Q'}{l ::^{\delta}_{\Sigma} \mathbf{eval}(Q)@l'.P \parallel l' ::^{\delta'}_{\Sigma'} P' \rightarrowtail l ::^{\delta}_{\Sigma} P \parallel l' ::^{\delta'}_{\Sigma'} P' \mid Q'}$
(In)	$\dfrac{match^{\delta}_{l}(\mathcal{T}[\![\, T \,]\!]_\delta, et) = \langle \delta'', \sigma \rangle}{l ::^{\delta} \mathbf{in}(T)@l'.P \parallel l' ::^{\delta'} \langle et \rangle \rightarrowtail l ::^{\delta[\delta'']} P\sigma \parallel l' ::^{\delta'} \mathbf{nil}}$
(Read)	$\dfrac{match^{\delta}_{l}(\mathcal{T}[\![\, T \,]\!]_\delta, et) = \langle \delta'', \sigma \rangle}{l ::^{\delta} \mathbf{read}(T)@l'.P \parallel l' ::^{\delta'} \langle et \rangle \rightarrowtail l ::^{\delta[\delta'']} P\sigma \parallel l' ::^{\delta'} \langle et \rangle}$
(New)	$\dfrac{l' \notin L \qquad \delta'[{}^{l'}\!/\!{}_{u}] \preceq \delta^{-n}[l' \mapsto \delta(l)] \qquad \delta'[{}^{l'}\!/\!{}_{u}] \vdash^{\Sigma}_{l'} \overline{\Sigma} \rhd \Sigma'}{L \vdash l ::^{\delta}_{\Sigma} \mathbf{newloc}(u : \delta').P \rightarrowtail L \cup \{l'\} \vdash l ::^{\delta[l' \mapsto (\delta(l) - \{n\})]}_{\Sigma} P[{}^{l'}\!/\!{}_{u}] \parallel l' ::^{\delta'[l'/u]}_{\Sigma'} \mathbf{nil}}$
(Call)	$l ::^{\delta}_{\Sigma} A \rightarrowtail l ::^{\delta}_{\Sigma} P \qquad \text{if } \Sigma = \Sigma' \cup \{A \triangleq P\}$
(Mark)	$\dfrac{l' = tgt(a) \qquad \delta(l') \sqsubseteq_{\Pi} \{cap(a)\} \qquad l ::^{\delta} a.P \parallel l' ::^{\delta'} Q \rightarrowtail N}{l ::^{\delta} \underline{a}.P \parallel l' ::^{\delta'} Q \rightarrowtail N}$
(Split)	$\dfrac{L \vdash l ::^{\delta} P \parallel l ::^{\delta} Q \parallel N \rightarrowtail L' \vdash l ::^{\delta'} P' \parallel l ::^{\delta''} Q' \parallel N'}{L \vdash l ::^{\delta} P \mid Q \parallel N \rightarrowtail L' \vdash l ::^{\delta'[\delta'']} P' \mid Q' \parallel N'}$
(Par)	$\dfrac{L \vdash N_1 \rightarrowtail L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \rightarrowtail L' \vdash N_1' \parallel N_2}$
(Struct)	$\dfrac{N \equiv N_1 \qquad L \vdash N_1 \rightarrowtail L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \rightarrowtail L' \vdash N'}$

Definition 2. *A net is* executable *if the following conditions hold:*

(i) for each node $l ::^{\delta}_{\Sigma} P$, with $\Sigma = \{A_1 \triangleq P_1, \ldots, A_n \triangleq P_n\}$, it holds that $\delta \vdash^{\Sigma}_{l} P \rhd P$ and $\delta \vdash^{\Sigma}_{l} P_i \rhd P_i$, for each $i \in \{1, ..., n\}$,

(ii) for any pair of nodes $l ::^{\delta}_{\Sigma} P$ and $l' ::^{\delta'}_{\Sigma'} P'$, it holds that $\overline{\Sigma} = \overline{\Sigma'}$,

where for inferring the type judgements, in addition to the rules in Table 2 and to axiom ($\star$) for $\langle et \rangle$, one can also use the rule

$$(\star\star) \qquad \frac{upd(\Gamma, arg(a)) \vdash^{\Sigma}_{l} P \rhd \underline{P}}{\Gamma \vdash^{\Sigma}_{l} \underline{a}.P \rhd \underline{a}.\underline{P}}$$

that allows a process to already contain marked actions.

Notice that executable nets are well-typed. Our main results will be stated in terms of executable nets; indeed, due to the dynamic acquisition of privileges, well-formed nets that are statically deemed well-typed can still give rise to run-time errors. However, by marking those actions that should be checked at run-time, well-typed (and well-formed) nets can be transformed into executable nets that, instead, cannot give rise to run-time errors (see Corollary 1).

It can be easily seen that the property of being executable is preserved by structural congruence. The following theorem states that it is also preserved by the reduction relation.

Theorem 1 (Subject Reduction). *If N is executable and $loc(N) \vdash N \rightarrowtail L' \vdash N'$ then N' is executable and $loc(N') \subseteq L'$.*

Now, we introduce the notion of *run-time error*, defined in terms of predicate $N \uparrow l$ that holds true when, within N, a process P running at node $l ::^{\Sigma}_{\delta} \_$ attempts to perform an action that is not allowed by δ or invokes a process that is not in Σ. The key rules are

$$\frac{\delta(tgt(a)) \not\sqsubseteq_{\Pi} \{cap(a)\}}{l ::^{\delta}_{\Sigma} a.P \uparrow l} \qquad \frac{\nexists \Sigma' \ : \ \Sigma = \Sigma' \cup \{A \triangleq P\}}{l ::^{\delta}_{\Sigma} A \uparrow l}$$

We can now state type safety, i.e. that executable nets do not give rise to run-time errors.

Theorem 2 (Type Safety). *If N is executable then $N \uparrow l$ for no $l \in loc(N)$.*

By combining together Theorem 1 and 2, and by denoting with $\rightarrowtail^*$ the reflexive and transitive closure of $\rightarrowtail$, we obtain the following result.

Corollary 1 (Global Type Soundness). *If N is executable and $loc(N) \vdash N \rightarrowtail^* L' \vdash N'$ then $N' \uparrow l$ for no $l \in loc(N')$.*

Type soundness is one of the main goal of any 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, where in general hosts are under the control of different authorities. 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 main result. To this aim, we define the *restriction* of a net N to a set of localities S, written N_S, as the subnet obtained from N by deleting all nodes whose addresses are not in S. The wanted local type soundness result can be formulated as follows.

Theorem 3 (Local Type Soundness). *Let N be a net and $S \subseteq loc(N)$. If N_S is executable and $loc(N) \vdash N \rightarrowtail^* L' \vdash N'$ then for no $l \in S$ it holds that $N' \uparrow l$.*

6 Example: Subscribing Online Publications

By means of a simple example, here we show the μKLAIM programming style and illustrate how to exploit its type system. For programming convenience, we will use the full version of the calculus [14], assume integers and strings to be basic values of the language, and omit trailing occurrences of process **nil** and the process defining equations.

Suppose that a user U wants to subscribe a 'licence' to enable accessing on-line publications by a given publisher P. To model this scenario we use three localities, l_U, l_P and l_C, respectively associated to U, P and to the repository containing P's on-line accessible publications. First of all, U sends a subscription request to P including its address (together with an 'out' capability) and credit card number; then, U waits for a tuple that will grant it the 'read' privilege needed to access P's publications and proceeds with the rest of its activity. The behaviour described so far is implemented by the process

$$U \triangleq \ \mathbf{out}(\text{“}\mathit{Subscr}\text{”}, l_U : [l_P \mapsto \{o\}], \mathit{CrCard})@l_P.\mathbf{in}(\text{“}\mathit{Acc}\text{”}, l_C : \{r\})@l_U.R$$

where process R may contain operations like $\mathbf{read}(\ldots)@l_C$. P, once it has received the subscription request and checked (by possibly using a third party authority) the validity of the payment information, gives U a 'read' capability over l_C. P's behaviour is modelled by the following process.

$$\begin{aligned} P \triangleq\ & \mathbf{in}(\text{“}\mathit{Subscr}\text{”}, !x : \{o\}, !y)@l_P.\ \textit{check credit card } y \textit{ of } x \textit{ and require the payment}\ . \\ & \mathbf{out}(\text{“}\mathit{Acc}\text{”}, l_C : [x \mapsto \{r\}])@x \ \mid\ P \end{aligned}$$

For processes U and P to behave in the expected way, the underlying net architecture, namely distribution of processes and security policies, must be appropriately configured. A suitable net is:

$$\begin{aligned} & l_U ::^{[l_U \mapsto \{o,i,r,e,n\},\, l_P \mapsto \{o\}]}\ \underline{U} \ \ \parallel\ \ l_P ::^{[l_P \mapsto \{o,i,r,e,n\}, l_C \mapsto \{o,i,r\}]}\ P \ \ \parallel \\ & l_C ::^{[\,]}\ \langle \mathit{paper}1 \rangle \mid \langle \mathit{paper}2 \rangle \mid \ldots \end{aligned}$$

where we have intentionally used $\underline{U}$ to emphasize the fact that the static type checking might have marked some actions occurring in U, e.g. the $\mathbf{read}(\ldots)@l_C$ actions in R. Upon completion of the protocol, the net will be

$$\begin{aligned} & l_U ::^{[l_U \mapsto \{o,i,r,e,n\}, l_P \mapsto \{o\}, l_C \mapsto \{r\}]}\ \underline{R} \ \ \parallel\ \ l_P ::^{[l_P \mapsto \{o,i,r,e,n\}, l_C \mapsto \{o,i,r\}, l_U \mapsto \{o\}]}\ P \ \ \parallel \\ & l_C ::^{[\,]}\ \langle \mathit{paper}1 \rangle \mid \langle \mathit{paper}2 \rangle \mid \ldots \end{aligned}$$

Notice that knowledge of address l_C is not enough for reading papers: the 'read' capability is needed. Indeed, security in the μKLAIM framework does not rely on name knowledge but on security policies. Moreover, once the 'read' capability over l_C has been acquired, all processes eventually spawned at l_U can access P's on-line publications. In other terms, U obtains a sort of 'site licence' valid for all processes running at l_U. This is different from [10], where, by using the same protocol, U would have obtained a sort of 'individual licence'. Notice also that the licence passed by P to U can be used only at l_U since the capability specification associated to l_C only grants l_U privilege r over l_C. Finally, no denial-of-service attack could be mounted through the access of tuple $\langle \text{“}\mathit{Acc}\text{”}, l_C : [l_U \mapsto \{r\}] \rangle$ located at l_U by processes running at sites of the network different from those explicitly mentioned because only processes running at l_U can retrieve the tuple (see rules (3) and (4) in Table 4).

Variants. We now touch upon a few variants (thoroughly presented in [14]) of the μKlaim framework and use the example for motivating their introduction. The variants differ in simple technical details and, mainly, in the burden charged to the static inference.

In real situations, a (mobile) process could dynamically acquire some privileges and, from time to time, decide whether it wants to keep them for itself or to share them with other processes running at the same node. In our example, U might just buy an 'individual licence'. The μKlaim framework can smoothly fit for this feature, by associating privileges also to processes and letting them decide whether an acquisition must enrich their hosting node or themselves. Moreover, the subscription could have an expiration date, e.g., it could be an annual subscription. Timing information can easily be accommodated in the μKlaim framework by simply assigning privileges a validity duration and by updating these information for taking into account time passing. 'Acquisition of privileges' can be thought of as 'purchase of services/goods'; hence it is natural that a process will lose the acquired privilege once it uses the service or passes the good to another process. In our running example, this corresponds to purchasing the right of accessing P's publications a given number of times. A simple modification of the μKlaim framework, for taking into account multiplicities of privileges and their consumption (due, e.g., to execution of the corresponding action or to cession of the privilege to another process), can permit to deal with this new scenario. Finally, the granter of a privilege could decide to revoke the privilege previously granted. In our example, P could prohibit U from accessing its publications because of, e.g., a misbehaviour or expiry of the subscription time (in fact, this is a way of managing expiration dates without assigning privileges a validity duration). Again, by annotating privileges dynamically acquired with the granter identity and enabling processes to use a new 'revoke' operation, the μKlaim framework can be extended to also manage privileges revocation.

7 Related Work

By now, there is a lot of work on type systems for security in calculi with process distribution and mobility; however, to the best of our knowledge, the type system we have presented in this paper is the first one that permits dynamic modification of security policies. We conclude by touching upon more strictly related work.

The research line closest to ours is that on the Dπ-calculus [16], a distributed version of the π–calculus equipped with a type system to control privileges of mobile processes over located communication channels. [15,20] present two improved type systems for the Dπ-calculus that, by relying on both local type information and on dynamic checking of incoming processes, permit establishing well-typedness of part of a network (like our local type soundness result). Like μKlaim, the Dπ-calculus relies on a flat network architecture; however, differently from μKlaim, the network infrastructure is not independent of the processes running over it and communication is local and channel-based. Moreover, node types describe permissions to use local channels. This is in sharp contrast with μKlaim types that aim at controlling the remote operations that a network node can perform over the other network nodes.

[23] presents D$\pi\lambda$, a process calculus resulting 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 not explicitly referrable by processes and just used to express process distribution. In [24], a fine-grained type system for D$\pi\lambda$ is defined that permits controlling the effect over local channels of transmitted processes parameterized w.r.t. channel names. Processes are assigned fine-grained types that, like interfaces, record the channels to which processes have access together with the corresponding capabilities, and parameterized processes are assigned dependent functional types that abstract from channel names and types. This use of types is akin to μKlaim one, though the differences between the underlying languages still remain.

Finally, we want to mention some proposals for the Mobile Ambients calculus and its variants, albeit their network models and mobility mechanisms are very different from those of μKlaim. Among the type systems more strictly related to security we recall those disciplining the types of the values exchanged in communications [5,4], those for controlling ambients mobility and ability to be opened [6,17,12,7] and that for controlling resources access via policies for mandatory access control based on ambients security levels [3].

Acknowledgements. We thank the anonymous referees for their useful comments.

References

1. K. Arnold, E. Freeman, and S. Hupfer. *JavaSpaces Principles, Patterns and Practice.* Addison-Wesley, 1999.
2. L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software – Practice and Experience*, 32:1365–1394, 2002.
3. M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *Proceedings of CONCUR 2001*, number 2154 in LNCS, pages 102–120. Springer, 2001.
4. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proceedings of TACS 2001*, number 2215 in LNCS, pages 38–63. Springer, 2001.
5. L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings of POPL '99*, pages 79–92. ACM, 1999.
6. L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the ambient calculus. *Information and Computation*, 177:160–194, 2002.
7. G. Castagna, G. Ghelli, and F. Z. Nardelli. Typing mobility in the seal calculus. In *Proceedings of CONCUR 2001*, number 2154 in LNCS, pages 82–101. Springer, 2001.
8. P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE TSE*, 24(5):362–366, 1998.
9. 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.

10. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
11. D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proceedings of ISADS 2001*, pages 278–286. IEEE, 2001.
12. M. Dezani-Ciancaglini and I. Salvo. Security types for mobile safe ambients. In *Proceedings of ASIAN'00*, volume 1961 of *LNCS*, pages 215–236, Springer, 2000.
13. D. Gelernter. Generative Communication in Linda. *Transactions on Programming Languages and Systems*, 7(1):80–112. ACM, 1985.
14. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2003. Available at `http://rap.dsi.unifi.it/~pugliese/DOWNLOAD/muklaim-full.pdf`.
15. M. Hennessy and J. Riely. Type-Safe Execution of Mobile Agents in Anonymous Networks. In *Secure Internet Programming*, volume 1603 of *LNCS*, pages 95–115. Springer, 1999.
16. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.
17. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL'00*, pages 352–364. ACM, 2000.
18. A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):251–269, 1999.
19. 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.
20. J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proc. of POPL'99*, pages 93–104. Full version to appear in *Journal of Automated Reasoning, 2003.*
21. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.
22. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998.
23. N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 557–572. Springer, 1999.
24. N. Yoshida and M. Hennessy. Assigning types to processes. In *Proceedings of LICS'00*, pages 334–348. Full version appear in Information and Computation, 173:82–120, 2002.