Journal of Logic and Algebraic Programming, 78:665 – 689. ©Elsevier, 2009.

Dynamic management of capabilities in a network aware coordination language☆

Daniele Gorla[a], Rosario Pugliese[b,*]

[a]*Dipartimento di Informatica, Università di Roma "La Sapienza"*
[b]*Dipartimento di Sistemi e Informatica, Università di Firenze*

Abstract

We introduce a capability-based access control model integrated into a linguistic formalism for modeling network aware systems and applications. Our access control model enables specification and dynamic modification of policies for controlling process activities (mobility of code and access to resources). We exploit a combination of static and dynamic checking and of in-lined reference monitoring to guarantee absence of run-time errors due to lack of capabilities. We illustrate the usefulness of our framework by using it for implementing a simplified but realistic scenario. Finally, we show how the model can be easily tailored for dealing with different forms of capability acquisition and loss, thus enabling different possible variations of access control policies.

Key words: Capability-based Access Control, Process Distribution and Mobility, Resource Usage, Mobility Control.

1. Introduction

In recent years, highly distributed networks have become a common infrastructure for many applications that exploit network facilities to access remote resources and services. These systems (e.g., the Internet) are highly *open*: their overall structure can change dynamically in unpredictable ways because the entities involved can join and leave the system at any time and need not be defined

☆This work is partially based on a preliminary paper appearing in [35] and has been partially supported by the EU project SENSORIA, IST-2005-016004.

*Corresponding author

Email addresses: gorla@di.uniroma1.it (Daniele Gorla), pugliese@dsi.unifi.it (Rosario Pugliese)

prior to starting the infrastructure. In developing applications for such computing environments, *network awareness* has emerged as a key design principle to deal with dynamic changes of network environments (e.g., variable guarantees for communication, cooperation, mobility, resource usage, security, etc.). Open network systems are then fostering the development of new paradigms and programming languages with mechanisms for handling process distribution and mobility, for coordinating process execution and interaction, and for managing resources and security. To improve the understanding of such complex mechanisms, several foundational process calculi (e.g. Distributed Join-calculus [29], Distributed π-calculus [41], Ambient calculus [14], and Seal calculus [15]) and process-based prototype languages (e.g. KLAIM [24], Lime [50] and Nomadic Pict [61]) have been devised.

Our study stems from the language KLAIM (*Kernel Language for Agents Interaction and Mobility*), an experimental language designed for network aware programming and implemented in Java [8]. In KLAIM, a system is a network of addressable nodes that contain running processes and data repositories. The nodes of a KLAIM net do not necessarily correspond to physically distributed machines; rather, they are units of abstraction for groups of processes and data, that belong to the same logical partition of a machine or to the same class of users. KLAIM communication model builds over, and extends, LINDA's notion of *generative communication* through tuple spaces [31]: processes may communicate asynchronously by exchanging tuples (i.e. sequences) of information items through tuple spaces (i.e. multisets of tuples); tuples are anonymous and retrieved from a tuple space by associative selection through a pattern-matching mechanism. The LINDA model, originally proposed for parallel programming on isolated machines, has later been extended with multiple, possibly distributed, tuple spaces [32] to improve modularity, scalability and performance. Indeed, the tuple space paradigm has become a popular alternative to (more conventional) point-to-point communication approaches; this is witnessed by the many tuple space based run-time systems, both from industries (e.g. SUN JavaSpaces [56, 5] and IBM T Spaces [63]) and from universities (e.g. PageSpace [21], WCL [52], KLAIM [24], Lime [50] and TuCSoN [49]).

Coordination, distribution and mobility of programs are important aspects for programming in open environments, but ensuring correct use of resources and data is crucial as well. Indeed, a host receiving mobile processes for execution needs tools to control access to its resources and to protect them from misuse by the incoming processes. Therefore, to prevent accidental or malicious manipulation of nodes' content, KLAIM has been equipped with a capability-based type system

[25]. Generally speaking, a *capability* is an unforgeable and tamper proof token given to a subject that specifies which kind of operations on a certain object are permitted to the holder of the capability. Subjects can be, e.g., mobile processes or network nodes; objects are resources like shared data, files, nodes, the network and so forth; access then means what kind of operations can be done on these resources (e.g., producing a datum, writing or reading a file, changing the network topology, moving about the network). In KLAIM, capabilities are used to specify the access control policies stating what operations (read, write, execute, ...) processes are allowed to perform while running at a given node; type checking then determines if processes comply with the policy of their hosting node. Hence, access requests are (mostly) checked statically, which is an advantage with respect to more traditional approaches to system security. The latters usually exploit a component called *reference monitor* that dynamically intercepts each attempted access to a resource; every acces is then processed by a combination of authentication (i.e., the identification of subject's identity) and authorization (i.e., the decision on whether the access should be allowed or denied).

Defining policies in terms of capabilities makes our approach well-suited for open systems, where capability-based approaches offer more flexibility than other protection mechanisms like, e.g., access control lists [60, 62, 20, 54, 45]. However, access policies in KLAIM [25] are fixed prior to starting system execution and cannot be flexibly modified according to its dynamic evolution; moreover, the type system presupposes a static knowledge of the entire system. These assumptions are unrealistic in open network systems. On one hand, access control information could be statically partial, inaccurate or missing: for example, a component may not initially have all the information it needs to authorize an access, or a requestor may not initially have all the necessary rights to access a resource. On the other hand, access control policies are likely to change once programs begin their execution, for example whenever new objects are created, or existing subjects leave the system or change duties.

To avoid illegal accesses to resources, dynamic modifications of access control information must be suitably managed. In this paper, we show that capabilities and interprocess communication can serve this purpose. In fact, mechanisms based on capabilities supplement the dinamicity inherent in open systems as they support introduction of user-defined rights and let subjects freely join and leave the system. Moreover, capabilities can be distributed and transferred by exploiting disciplined communication operations. In this way, we can increase the flexibility of the original KLAIM capability-based access control model with the possibility of dynamically changing policies.

To draw attention on the key aspects, we leave out from KLAIM some linguistic features that would complicate the technical treatment of the protection model more than necessary; we call the resulting language μKLAIM (*micro* KLAIM), since it can be thought of as the process calculus on which KLAIM is based. In the resulting framework, subjects and objects are both network nodes; this corresponds to the fact that the initiator of one operation can be the target of another. Moreover, instead of directly performing an action, a process can use a capability to delegate another node the ability to perform that action. In our setting, this passing of access rights is implemented by exploiting interprocess communication primitives, thus providing means for controlling exchange of rights. In practice, when a node address is exchanged in a communication, a capability on that node is passed in order to grant the receiver a set of access rights on that node. Access control policies are then susceptible of dynamic modifications due to, e.g., capability acquisition or consumption. Of course, capabilities are protected from forgery: the only way for nodes and processes to obtain capabilities is to have them granted at the outset or as result of some communication.

Execution monitoring together with mechanisms supporting modifications at run-time of access control policies turn out to be essential for dealing with network-aware applications such as, e.g., resource discovery and e-commerce. However, like for the KLAIM type system, we maintain a static checking phase to reduce run-time checks and improve system performance. In fact, we interpret access control policies as process types and develop a sort of type checking procedure that statically checks process intentions against the local access control policy. A similar procedure is then used at run-time to check compatibility of the intentions of a migrating process with the access policy of the destination node: only if this check succeeds, the migrating process is sent for execution. This enhances the performance of the security monitor, but at the cost of dynamically checking the migrating process before entrance; a further optimization could be obtained by exploiting forms of 'proof carrying code' [46]. Our checking mechanisms ensure a safety property that is preserved along system evolutions: every checked node is 'safe' in the sense that no process, while running at it, will ever attempt to perform an operation which is not authorized by the local policy. This result fits well with the key features of open systems, where 'good' components usually run in hostile environments.

This paper does not cover many issues in the area. We only consider authorization mechanisms based on capabilities and discretionary access control policies (i.e. policies based on the identity of the subject attempting the access). However, we do not commit ourself to any specific policy; we model the mechanisms needed

to enforce them, not their initial setting up. Of the underlying distributed computing base, we assume that shared-key encryption and/or public-key encryption are available where needed, e.g. to support scalable authentication and authorization protocols. For example, an identity and some credentials, which represent statements certified by given entities (e.g., certification authorities), can be associated with every component in a system; credentials are then used to prove the component's identity to all other components (e.g., [23]) or can also be directly bound to authorizations (see, e.g., PolicyMaker [10], Keynote [9], REFEREE [19], DL [43]). Cryptographic mechanisms (e.g., one-way functions like in Amoeba [58] and in ICAP [33]) can also be exploited to prevent processes from forging new capabilities or tampering with existing ones. Finally, we also assume that node's run-time is reliable; in particular, its reference monitor is a tamper-proof, non-bypassable, trusted component intercepting each and every attempted access to a system and its resources.

The rest of the paper is organized as follows. Section 2 formally describes the syntax of μKLAIM and presents an example application inspired by a realistic publisher/subscriber scenario. The static semantics of the language is in Section 3 and its dynamic semantics is in Section 4. Section 5 proves the correctness of our framework. Section 6 gives a full account of the example described in Section 2. Section 7 presents some variations of the access control model where, e.g., capabilities can be revoked, expire or whose distribution can be controlled. Section 8 contains comparisons with related work, and Section 9 concludes the paper by also arguing on a few language design issues. Appendix A reports the proofs of some technical results, while Appendix B reports the formal definitions for the variations of Section 7.

Our presentation is incremental: the basic framework presented and discussed until Section 6 is intentionally simplistic, since it misses several desirable and expectable features. The framework is then enhanced in Section 7, where more complex features are added. We find it useful to present our approach step-by-step, to clarify its main issues without hiding them behind heavy notations: we start from a collaborative framework, well-suited for intranets, and then gradually move to a more complex and realistic framework, closer to open nets.

2. The Language μKLAIM

μKLAIM (*micro* KLAIM), is a minimal variation of KLAIM that still retains all KLAIM's distinctive features: explicit distribution of processes and data, remote operations, process mobility and asynchronous communication through multiple

NETS:
$N \quad ::= \quad l ::^{\delta} C \;\mid\; N_1 \parallel N_2$

NODE COMPONENTS:
$C \quad ::= \quad \langle t \rangle \;\mid\; P \;\mid\; C_1 \mid C_2$

PROCESSES:
$P \quad ::= \quad \mathbf{nil} \;\mid\; a.P \;\mid\; \underline{a}.P \;\mid\; P_1 \mid P_2 \;\mid\; * P$

PROCESS ACTIONS:
$a \quad ::= \quad \mathbf{in}(T)@u \;\mid\; \mathbf{read}(T)@u \;\mid\; \mathbf{out}(t)@u \;\mid\; \mathbf{eval}(P)@u \;\mid\; \mathbf{newloc}(l : \delta)$

TEMPLATES:
$T \quad ::= \quad u \;\mid\; !\, x : \pi \;\mid\; T_1, T_2$

TUPLES:
$t \quad ::= \quad u : \mu \;\mid\; t_1, t_2$

Table 1: μKLAIM syntax

distributed data repositories. With respect to KLAIM, μKLAIM has a simpler syntax and operational semantics without higher-order communication, without allocation environments for translating one kind of addresses into the other (in fact, μKLAIM has only one kind of node addresses) and with replication in place of parameterized process definitions.

The syntax of μKLAIM is reported in Table 1. The use of information enabling access control (i.e. *sets of access rights* π, *capability lists* δ, and *grantings* μ) and of the highlighted constructs (which do not occur in source terms) will be explained in Section 3. We assume a countable set $\mathcal{N}$ of *names* $l, l', \ldots, u, \ldots, x, y, \ldots$, that can be used as localities or variables. Notationally, we prefer letters $l, l', \ldots$ when we want to stress the use of a name as a locality and $x, y, \ldots$ when we want to stress the use of a name as a variable. We will use u for a generic name.

Tuples are sequences of names, each associated to a granting. *Templates* are patterns used to select tuples in tuple spaces. They are sequences of names and formal fields; the latter ones are written $!\, x : \pi$ and are used to bind names. *Processes* are the μKLAIM active computational units. They are built up from the inert process **nil** and from five basic operations, called *actions*, by using action prefix-

ing, parallel composition and replication. Actions can be marked (i.e. underlined) to charge the reference monitor with the run-time check for availability of the needed capabilities. The informal semantics of process actions is as follows. Action $\mathbf{in}(T)@u$ looks for a matching tuple t in the tuple space (TS, for short) located at u. Intuitively, a template matches against a tuple if both have the same number of fields and corresponding fields match; this happens if both fields are the same name or one is a formal and the other one is a name. If a matching t is found, it is removed from the TS, the formal fields of T are replaced in the continuation process with the corresponding names of t and the operation terminates; otherwise, the operation is suspended until a matching tuple becomes available. Action $\mathbf{read}(T)@u$ is similar but it does not remove the selected tuple from the TS. Action $\mathbf{out}(t)@u$ adds the tuple t to the TS located at u. Action $\mathbf{eval}(P)@u$ sends process P for execution at u. Action $\mathbf{newloc}(l : \delta)$ dynamically creates a new network node with address l and capability list δ. Notice that **newloc** is the only action not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take place.

Nets are finite collections of nodes where processes and tuple spaces can be allocated. A *node* is a triple $l ::^{\delta} C$, where locality l is the address of the node, C is the (parallel) component located at l and δ is the policy of the node. *Components* can be processes or (located) tuples. *Located tuples* (ranged over by $\langle t \rangle$) are inactive components representing tuples in a TS that have been inserted along a computation by executing an action **out**[1]. The TS located at l results from the parallel composition of all tuples residing at l.

Names occurring in process terms can be *bound* by action prefixes. More precisely, in processes $\mathbf{in}(T)@u.P$ and $\mathbf{read}(T)@u.P$ the prefixes bind the names in the formal fields of T, while in process $\mathbf{newloc}(l : \delta).P$ the prefix binds l. In all these cases, P is the scope of the bindings. A name that is not bound is called *free*. The sets $\text{BN}(P)$ and $\text{FN}(P)$ (of bound and free names, resp., of P) are defined accordingly, and so is *alpha-conversion*, denoted $=_{\alpha}$. With abouse of notation, we shall extend $\text{BN}(\cdot)$ and $\text{FN}(\cdot)$ to nets, with the expected meaning.

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. The laws say that $\|$ is commutative and associative, that alpha-convertible processes are interchangeable and that process **nil**

[1]Like in [24], here we are assuming that at the outset no tuple is present in the existing TSs. Section 3 will clarify why this simplifying assumption is useful.

(Com)	$N_1 \parallel N_2 \equiv N_2 \parallel N_1$	(Ass)	$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$
(Alpha)	$\dfrac{P =_\alpha P'}{l ::^\delta P \equiv l ::^\delta P'}$	(Abs)	$l ::^\delta C \equiv l ::^\delta (C\vert\mathbf{nil})$

Table 2: Nets structural congruence

can be absorbed/spawned. 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).

To sketch the access control model integrated in the language, we now present a simplified but realistic publisher/subscriber scenario where our framework turns out to be expressive and elegant; this informal presentation will be refined in Section 6.

Example 2.1 Let l_U be the address of a node representing the server of a given department and let l_P be the address of a node representing the publisher of some on-line publications that are stored at address l_S. We want to implement a protocol through which the head of department subscribes a 'license' enabling all the department members to access the publications at l_S. In terms of access control, this means that the protocol must extend the policy of node l_U (expressing the operations that processes hosted at l_U are allowed to perform over the net) with the capability of reading papers from l_S. Hence, if the department server starts with policy δ, upon completion of the protocol l_U's policy should become $\delta[l_S \mapsto \{r\}]$; this notation means that processes at l_U can read tuples from l_S's tuple space, while still being enabled to perform those actions enabled by δ. Then, a department member located at l_M can spawn code over l_U and retrieve l_P's papers by simply using the process

$$\mathbf{eval}\big(\,\mathbf{read}(paperTitle, !x)@l_S.\mathbf{out}(paperTitle, x)@l_M\,\big)@l_U$$

Action $\mathbf{eval}(P)@l_U$ spawns code P for execution at l_U. Then, action $\mathbf{read}(paperTitle, !x)@l_S$ looks for a tuple matching the template $paperTitle, !x$ (i.e. a paper whose title is $paperTitle$ and whose body is a text B); if such a paper is found, x is replaced by B in the continuation process. Finally, action $\mathbf{out}(paperTitle, B)@l_M$ inserts in l_M's tuple space a tuple containing the paper

required by the department member whose address is l_M. In a more realistic scenario, the capability 'read' over l_S will not be delivered forever to l_U. This scenario could be modeled by exploiting some of the variations described in Section 7.

3. Static Semantics

Informally, for each node of a net, say $l ::^{\delta} C$, the task of the access control system is to determine if the actions that C intends to perform when running at l are enabled by the access policy δ. When asking about authorization of a particular action, there are typically three possible outcomes: 'yes', the action may be executed because sufficient access rights exist in δ for the action to be approved; 'no', the action may not be executed because sufficient access rights in δ do not exist and cannot be acquired; 'unknown', sufficient access rights to approve the action do not exist in δ, but they could be dynamically acquired and the decision about authorization is delayed at run-time. In the latter case, run-time checks are unavoidable and are charged to the reference monitor (see rule (MARK) in Section 4). In our opinion, the crucial novelty of our approach is the integration of the 'unknown' possibility in the static checking. Indeed, all the static analysis techniques we are aware of either accept or reject a system/process; instead, the combination of static checking and of in-lined monitoring we propose allows us to deal with dynamic policy modifications without compromising system performance too much.

3.1. Access Rights, Capabilities, Capability Lists and Grantings

In the previous Section, we have seen that access control information occurs in the syntax. Below, we briefly outline how these information are exploited. First, each name x occurring in a formal field of the template specified as argument of an action **read/in** is explicitly associated[2] to a *set of access rights* π; these are the rights necessary to the continuation process to perform its operations on x while running locally. Second, in actions **out** each name in the spawned tuple is associated to a (possibly empty) *granting* μ that specifies the capabilities passed through along with that name. Third, each node $l ::^{\delta} C$ is equipped with a *capability list* δ describing its access policy. Similarly, when nodes are dynamically created by actions **newloc**, a capability list is used to specify their access policy.

[2]Such a set is not strictly necessary: it could be inferred by examining how the continuation process uses x. However, its presence enables a simpler static checking.

Definition 3.1. *The set of* access rights*, C, is $\{r, i, o, e\}$ and is ranged over by c. We let Π be the powerset of C and use π to range over Π.* Capabilities *are pairs made by a locality l and a set of access rights π, written $l \mapsto \pi$.* Capability lists*, ranged over by δ, and* grantings*, ranged over by μ, are finite partial functions mapping $\mathcal{N}$ to Π.*

We use r, i, o and e to indicate the operation whose name begins with it. For example, e is used to control process mobility; thus, the capability $l' \mapsto \{e\}$ in the policy of locality l enables processes running at l to perform actions **eval** over l'. Differently from previous presentations, we do not use any capability to control actions **newloc**: to simplify notation, here we assume that they are always enabled.

Notationally, a capability list mapping l_i to a non-empty π_i, for $i = 1, \ldots, k$, will be written as $[l_i \mapsto \pi_i]_{i=1,\ldots,k}$; a similar notation is exploited also for grantings but with a different meaning. Indeed, grantings are used in actions **out** to specify the capabilities to be passed through along with a node address. For example, if a process running at l retrieves a tuple $\langle l' : \mu \rangle$, then the policy of l in enriched with the capability $l' \mapsto \mu(l)$; the latter allows processes running at l to perform at l' those actions whose rights are in $\mu(l)$.

We now introduce an ordering relation over capability lists that formalizes the property that a policy is more restrictive than another one. To this aim, we start with defining an ordering over sets of access rights that will induce the desired ordering on capability lists.

Definition 3.2. *$\pi_1 \sqsubseteq_\Pi \pi_2$ if and only if $\pi_1 \subseteq \pi_2$.*

Thus, if $\pi_1 \sqsubseteq_\Pi \pi_2$ then π_2 enables at least the actions enabled by π_1. However, the model we develop is completely parametric with respect to the used ordering over access rights and other alternatives are possible (see, e.g., [25] or Section 7).

By taking advantage of the fact that capability lists are partial functions, we exploit the standard pointwise union of partial functions to extend δ_1 with δ_2, written $\delta_1[\delta_2]$; this is the capability list δ with domain $dom(\delta_1) \cup dom(\delta_2)$ such that

$$\delta(u) \triangleq \begin{cases} \delta_1(u) & \text{if } u \in dom(\delta_1) - dom(\delta_2) \\ \delta_2(u) & \text{if } u \in dom(\delta_2) - dom(\delta_1) \\ \delta_1(u) \cup \delta_2(u) & \text{if } u \in dom(\delta_1) \cap dom(\delta_2) \end{cases}$$

Similarly, we exploit the standard pointwise inclusion of partial functions to order capability lists.

Definition 3.3 (Ordering). *We say that* δ_1 *is* less than δ_2 *(or that* δ_2 *is* greater than δ_1*), written* $\delta_1 \preceq \delta_2$*, if* $\delta_1(l) \sqsubseteq_\Pi \delta_2(l)$ *for every* $l \in dom(\delta_1)$.

The ordering $\preceq$ formalizes the idea that, if $\delta_1 \preceq \delta_2$, then δ_1 is a less permissive policy than δ_2. Clearly, $\preceq$ is decidable because we work with finite partial functions.

3.2. A Capability-based Access Control System

The task of the static phase is to lighten the need of run-time checks as much as possible; this will be done by exploiting all the security information occurring in the syntax of a μKLAIM net. There are however checks that must be deferred at run-time. First of all, compliance between the access control information in a tuple and that in the argument of an **in/read** can only be checked when executing the action. As explained in Section 2, an action **in/read** succeeds only if the template it specifies, T, matches against the accessed tuple, t. Thus, we charge the matching function (that is naturally present in any communication based on tuple spaces) with the burden of verifying access control compliance between T and t. Notice that this is the only technical commonality between our approach and the type system in [25].

Furthermore, capabilities are dynamically acquired as a result of executing a **read/in** and they are used to enrich the policy of the node where the action was fired. These capabilities are exploitable by all co-located processes, and not only by the process that performed the action. This choice has been driven by the principles underlying the notion of capabilities, where rights are assigned to subjects that, in our framework, are network nodes. Unluckily, to make dynamic acquisition meaningful, we need to introduce further run-time checks, because an action that is statically illegal could become legal upon acquisition of the capability enabling it. In such cases, the static access control mechanism simply *marks* (i.e. underlines) the action to require its checking at run-time by the reference monitor. This explains the use of the construct $\underline{a}.P$ in Table 1, where action a that prefixes process P is underlined. Notationally, we will write $\underline{P}$ ($\underline{C}$ and $\underline{N}$, resp.) to emphasise that process P (component C and net N, resp.) may contain marked actions.

The marking mechanism never applies to actions whose targets are names bound by **in/read**, because such actions can be statically checked. For example, our system has to reject node

$$l_1 ::^{[l' \mapsto \{r\}]} \mathbf{read}(!x : \{o\})@l'.\mathbf{read}(!y)@x$$

because r does not belong to the annotation of x, while it has to accept node

$$l_2 ::^{[l' \mapsto \{r\}]} \mathbf{read}(!x : \{o\})@l'.\mathbf{out}(t)@l'$$

because action $\mathbf{out}(t)@l'$ can be marked and checked at run-time. In fact, if x is dynamically replaced with l', l_2 will acquire the access right o over l' and the process running at l_2 can proceed; otherwise, the process will be suspended. In our system, the dynamic acquisition of capabilities is exploited exactly for relaxing the static checking and admitting nodes like l_2 while requiring on (part of) them a run-time checking.

Finally, when performing actions **out**, the grantings occurring within the argument must be checked. This is necessary to avoid capability forging like in

$$l ::^{\delta} \mathbf{out}(l' : [l \mapsto \pi])@l.\mathbf{in}(!x : \pi)@l$$

where $[l' \mapsto \pi] \not\leq \delta$. If the first action was legal, the second action would add new capabilities to δ and l would enlarge its policy autonomously. To avoid this access control breach, we must ensure that action **out** is executed only if $\pi \sqsubseteq_{\Pi} \delta(l')$. If this check was performed statically, then dynamically acquired capabilities could not be passed any longer and would be dealt with differently from those statically owned; this somehow collides with discretionary access control policies, where a dynamically received capability becomes a first-class capability (that must be handled like statically assigned ones).

We now formally define the static checking. It is defined in terms of *judgments* for components of the form $\Gamma \vdash_l^L C \triangleright \underline{C}$. Here, L is a finite set of names and it is used to keep track of bound names that have been freed during the inference as the result of removing a binding operator, i.e. **in**/**read**/**newloc**; this information will be used by the inference to determine if a given action must be marked. The *context* Γ is a capability list that collects together the capabilities contained in the policy of l and the annotations for the names that have been freed in C. Intuitively, the judgment $\Gamma \vdash_l^L C \triangleright \underline{C}$ states that, when C is located at l, the unmarked actions in $\underline{C}$ are admissible w.r.t. Γ. Instead, the marked actions in $\underline{C}$ cannot be deemed legal at compile time but could become permissible at run-time, after dynamic acquisition of the necessary capabilities (via execution of actions **in**/**read** performed at l). When L is empty, we shall simply write $\Gamma \vdash_l C \triangleright \underline{C}$.

To update a context with the sets of access rights specified within a template, we use the auxiliary function *upd* that behaves like the identity function for all fields but for template formal fields. Formally, it is defined by:

$$upd(\Gamma, T) \triangleq \begin{cases} upd(upd(\Gamma, T_1), T_2) & \text{if } T = T_1, T_2 \\ \Gamma \uplus [x \mapsto \pi] & \text{if } T = !\, x : \pi, \\ \Gamma & \text{otherwise} \end{cases}$$

Here, notation $\delta_1 \uplus \delta_2$ denotes pointwise union of partial functions with disjoint domains.

Notation 3.4 Given an action a different from **newloc**, we use $arg(a)$ to denote its argument, $tgt(a)$ its target location and $ar(a)$ the access right corresponding to a. For example, if a is **out**$(t)@l$, then we have $arg(a) \triangleq t$, $tgt(a) \triangleq l$ and $ar(a) \triangleq o$.

Judgments are inferred by using the rules in Table 3. The function $mark^L_\Gamma(\cdot)$ for marking process actions is defined as follows

$$mark^L_\Gamma(a) \triangleq \begin{cases} a & \text{if } \{ar(a)\} \sqsubseteq_\Pi \Gamma(tgt(a)) \\ \underline{a} & \text{if } \{ar(a)\} \not\sqsubseteq_\Pi \Gamma(tgt(a)) \text{ and } tgt(a) \notin L \end{cases}$$

where $\not\sqsubseteq_\Pi$ denotes the negation of $\sqsubseteq_\Pi$. Condition $tgt(a) \notin L$ distinguishes actions using localities as target from those using freed names, marking the former ones and rejecting the latter ones (as previously explained).

The rules in Table 3 should be quite explicative; we only remark a few points. Rule (T-DAT) says that located tuples always successfully pass the static checking, regardless their contents. This choice simplifies the technical development; however, to check grantings therein, we require that no tuple be present in the net at the outset (data must all be produced via actions **out**, that are dynamically checked). Rule (T-PAR) deals both with process composition and with component composition, while rule (T-REPL) deals with replication. Rule (T-SND) deals with **out** and **eval**; notice that checking the arguments of these actions is deferred at run-time. Rule (T-RCV) deals with **in** and **read**; the annotations in the formal fields of the template are used to enrich the current context in order to check the continuation process. Rules (T-MSND) and (T-MRCV) are similar to rules (T-SND) and (T-RCV), respectively, but allow a process to already contain marked actions. Action **newloc** is dealt with differently from the other actions by rule (T-NEW). Recall that it is always performed locally and that, for the sake of simplicity, we assume it is always enabled. However, to actually enable the creation, the specified access policy δ must be in agreement with the access policy of the node executing the operation extended with the ability of performing over l' all the operations allowed locally. This is needed to prevent a malicious node from forging capabilities by creating a

(T-NIL)
$\Gamma \vdash_l^L \mathbf{nil} \triangleright \mathbf{nil}$

(T-DAT)
$\Gamma \vdash_l^L \langle t \rangle \triangleright \langle t \rangle$

(T-PAR)
$$\frac{\Gamma \vdash_l^L C_1 \triangleright \underline{C_1} \qquad \Gamma \vdash_l^L C_2 \triangleright \underline{C_2}}{\Gamma \vdash_l^L C_1 \mid C_2 \triangleright \underline{C_1} \mid \underline{C_2}}$$

(T-REPL)
$$\frac{\Gamma \vdash_l^L P \triangleright \underline{P}}{\Gamma \vdash_l^L * P \triangleright * \underline{P}}$$

(T-SND)
$$\frac{ar(a) \in \{o, e\} \qquad \Gamma \vdash_l^L P \triangleright \underline{P}}{\Gamma \vdash_l^L a.P \triangleright mark_\Gamma^L(a).\underline{P}}$$

(T-MSND)
$$\frac{ar(a) \in \{o, e\} \qquad \Gamma \vdash_l^L P \triangleright \underline{P}}{\Gamma \vdash_l^L \underline{a}.P \triangleright \underline{a}.\underline{P}}$$

(T-RCV)
$$\frac{ar(a) \in \{i, r\} \qquad upd(\Gamma, arg(a)) \vdash_l^{L \cup \mathbf{BN}(arg(a))} P \triangleright \underline{P}}{\Gamma \vdash_l^L a.P \triangleright mark_\Gamma^L(a).\underline{P}}$$

(T-MRCV)
$$\frac{ar(a) \in \{i, r\} \qquad upd(\Gamma, arg(a)) \vdash_l^{L \cup \mathbf{BN}(arg(a))} P \triangleright \underline{P}}{\Gamma \vdash_l^L \underline{a}.P \triangleright \underline{a}.\underline{P}}$$

(T-NEW)
$$\frac{\delta \preceq \Gamma \uplus [l' \mapsto \Gamma(l)] \qquad \Gamma \uplus [l' \mapsto \Gamma(l)] \vdash_l^{L \cup \{l'\}} P \triangleright \underline{P}}{\Gamma \vdash_l \mathbf{newloc}(l' : \delta).P \triangleright \mathbf{newloc}(l' : \delta).\underline{P}}$$

Table 3: Static access control mechanism

new node with more powerful capabilities where sending a process that takes advantage of the capabilities not owned by the creator. Notice also that the creating node is assumed to have over the created one all the capabilities it owns on itself.

We now state an important property of the inference system of Table 3, namely that it is decidable. Its proof is given in Appendix A.

Proposition 3.5 (Decidability). *For any Γ, L, l, C and C' it is decidable to determine whether the judgment $\Gamma \vdash_l^L C \triangleright C'$ holds true or not.*

The proof of Proposition 3.5 is constructive because it also gives an algorithm

$$\frac{\mu = [l_i \mapsto \pi_i]_{i=1,\ldots,k} \qquad \forall\, i = 1, \ldots, k \,.\, \pi_i \sqsubseteq_{\Pi} \delta(l)}{[\![\, l : \mu \,]\!]_\delta} \qquad \frac{[\![\, t_1 \,]\!]_\delta \qquad [\![\, t_2 \,]\!]_\delta}{[\![\, t_1, t_2 \,]\!]_\delta}$$

Table 4: Rules to check grantings

that, for any Γ, L, l and C determines the C' with the smallest number of marked actions such that the judgment $\Gamma \vdash_l^L C \triangleright C'$ holds. The complexity of the algorithm is linear in the number of operators in C.

We will deem *admissible* those nets for which the static inference mechanism successfully terminate, as defined below.

Definition 3.6. *A net is* admissible *if, for each node* $l ::^\delta C$*, there exists a component* $\underline{C}$ *such that* $\delta \vdash_l C \triangleright \underline{C}$.

4. Dynamic Semantics

The first ingredient we need for defining the operational semantics is a mechanism to control the capabilities passed through while executing an action **out** from node l'. This check is defined as the predicate $[\![\, \cdot \,]\!]_\delta$, that can be inferred by using the rules in Table 4. $[\![\, \cdot \,]\!]_\delta$ is parameterized with respect to δ, the policy of the node l' where the action **out** takes place. Intuitively, whenever a tuple passes the access rights π_i over l to l_i (thus, the tuple is of the form $\langle l : [l_i \mapsto \pi_i] \rangle$), we need to verify that l' owns π_i.

Another ingredient we need is a formal way to say that a template and a tuple do match. The *pattern-matching* function, $match_l^\delta$, is defined by the rules in Table 5; it is parameterized with the locality l and the access control policy δ of the node where it is invoked. A successful matching returns a capability list, used to extend the policy δ of the node l with the capabilities delivered by the tuple, and a substitution, used to assign names to variables in the process invoking the matching. We use σ to range over substitutions (with finite domain) of names for names, ϵ to denote the 'empty' substitution and $\circ$ to denote substitutions composition. As usual, substitution application may require alpha-conversion to avoid capturing of free names.

Notice that the node where the **read/in** is executed must be authorized to access all the names occurring in the selected tuple; this is explicitly required in the

$$(\mathrm{M}_1)\quad \frac{l \in dom(\mu)}{match_l^\delta(l', l' : \mu) = \langle [\,], \epsilon \rangle} \qquad (\mathrm{M}_2)\quad \frac{\pi \sqsubseteq_\Pi \delta(l') \cup \mu(l)}{match_l^\delta(!\, x : \pi, l' : \mu) = \langle [l' \mapsto \pi], [l'/x] \rangle}$$

$$(\mathrm{M}_3)\quad \frac{match_l^\delta(T_1, t_1) = \langle \delta_1, \sigma_1 \rangle \qquad match_l^\delta(T_2, t_2) = \langle \delta_2, \sigma_2 \rangle}{match_l^\delta(\ (T_1, T_2)\ ,\ (t_1, t_2)\ ) = \langle \delta_1[\delta_2], \sigma_1 \circ \sigma_2 \rangle}$$

Table 5: Matching rules

premise of rule (M_1) and implicitly required by the fact that the $\mu(l)$ in the premise of rule (M_2) must be defined. This feature constraints the nodes from where tuples can be accessed (see Section 6). Moreover, rule (M_2) ensures that a formal field can be replaced by a locality l' only if π is enabled by the union of the access rights over l' owned by l and of the access rights over l' delivered to l by the tuple. The capabilities delivered by the tuple are then used to enrich the capabilities of l over l'.

Function $match_l^\delta$ satisfies the following property, whose proof can be easily done by induction on the number of fields of the first argument of the function.

Proposition 4.1. *If $match_l^\delta(T, t) = \langle \delta', \sigma \rangle$ with $dom(\sigma) = \{x_i\}_{i \in I}$, then $\delta' = [l_i \mapsto \pi_i]_{i \in I}$ where, for every $i \in I$, $!\, x_i : \pi_i$ is a field of T, $l_i : \mu_i$ is the corresponding field of t and $\sigma(x_i) = l_i$.*

As we already said, the operational semantics relates μKLAIM nets that may contain evaluated tuples and marked actions. It is given by a reduction relation, $\succ\!\!\rightarrow$, which is the least relation induced by the rules in Table 6. Net reductions are defined over configurations of the form $L \triangleright N$, where L is such that $\mathrm{FN}(N) \subseteq L \subset_{\mathrm{fin}} \mathcal{N}$. In a configuration $L \triangleright N$, L keeps track of the names occurring 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 $N \succ\!\!\rightarrow N'$ instead of $L \triangleright N \succ\!\!\rightarrow L \triangleright N'$.

Let us comment on the rules in Table 6. Rule (OUT) says that, before adding a tuple to a TS, the grantings within the tuple must be checked according to the policy δ of the node where the action is performed. Rule (EVAL) says that a process is allowed to migrate only if it complies with the access policy of the target node. During this preliminary check, some process actions could be marked to be effectively checked before execution. Rules (IN) and (READ) say that the process

(Out) $$\frac{[\![\ t\ ]\!]_\delta}{l ::^\delta \mathbf{out}(t)@l'.P \parallel l' ::^{\delta'} C' \ \succ\!\!\rightarrow\ l ::^\delta P \parallel l' ::^{\delta'} C'|\langle t\rangle}$$

(Eval) $$\frac{\delta' \vdash_{l'} Q \triangleright \underline{Q}}{l ::^\delta \mathbf{eval}(Q)@l'.P \parallel l' ::^{\delta'} C' \ \succ\!\!\rightarrow\ l ::^\delta P \parallel l' ::^{\delta'} C'|\underline{Q}}$$

(In) $$\frac{match_l^\delta(T,t) = \langle \delta'', \sigma\rangle}{l ::^\delta \mathbf{in}(T)@l'.P \parallel l' ::^{\delta'} \langle t\rangle \ \succ\!\!\rightarrow\ l ::^{\delta[\delta'']} P\sigma \parallel l' ::^{\delta'} \mathbf{nil}}$$

(Read) $$\frac{match_l^\delta(T,t) = \langle \delta'', \sigma\rangle}{l ::^\delta \mathbf{read}(T)@l'.P \parallel l' ::^{\delta'} \langle t\rangle \ \succ\!\!\rightarrow\ l ::^{\delta[\delta'']} P\sigma \parallel l' ::^{\delta'} \langle t\rangle}$$

(New) $$\frac{l' \notin L}{L \triangleright l ::^\delta \mathbf{newloc}(l' : \delta').P \ \succ\!\!\rightarrow\ L \cup \{l'\} \triangleright l ::^{\delta[l' \mapsto \delta(l)]} P \parallel l' ::^{\delta'} \mathbf{nil}}$$

(Repl) $$l ::^\delta *P \ \succ\!\!\rightarrow\ l ::^\delta P \mid *P$$

(Mark) $$\frac{l' = tgt(a) \qquad \{ar(a)\} \sqsubseteq_\Pi \delta(l') \qquad l ::^\delta a.P \parallel l' ::^{\delta'} C' \ \succ\!\!\rightarrow\ N}{l ::^\delta \underline{a}.P \parallel l' ::^{\delta'} C' \ \succ\!\!\rightarrow\ N}$$

(Split) $$\frac{L \triangleright l ::^\delta C_1 \parallel l ::^\delta C_2 \parallel N \ \succ\!\!\rightarrow\ L' \triangleright l ::^{\delta'} C_1' \parallel l ::^\delta C_2' \parallel N'}{L \triangleright l ::^\delta C_1|C_2 \parallel N \ \succ\!\!\rightarrow\ L' \triangleright l ::^{\delta'} C_1'|C_2' \parallel N'}$$

(Par) $$\frac{L \triangleright N_1 \ \succ\!\!\rightarrow\ L' \triangleright N_1'}{L \triangleright N_1 \parallel N_2 \ \succ\!\!\rightarrow\ L' \triangleright N_1' \parallel N_2}$$

(Struct) $$\frac{N \equiv N_1 \qquad L \triangleright N_1 \ \succ\!\!\rightarrow\ L' \triangleright N_2 \qquad N_2 \equiv N'}{L \triangleright N \ \succ\!\!\rightarrow\ L' \triangleright N'}$$

Table 6: μKLAIM operational semantics

performing the operation can proceed only if pattern-matching succeeds. In this case, the access policy of the receiving node is enriched with the capability list returned by the matching mechanism and the substitution returned along with the capability list is applied to the continuation of the process performing the operation (and in the annotations therein). In rule (New), the set L of localities already

in use is exploited to verify that l' is a fresh address. Notice that the policy of the creator is properly updated and the address of the new node is not initially known to any other node in the net; thus, l' can be used by the creating process as a sort of *private* resource (that, of course, can be later communicated to other processes). Rule (Repl) says that copies of a replicated process can be freely spawned. Rule (Mark) says that the in-lined reference monitor stops execution whenever the capability for executing a is missing. Rule (Split) transforms a parallel over components into a parallel over net nodes[3]. Rules (Par) and (Struct) are standard: the former says that, if part of a composed net evolves, the whole net evolves accordingly and the latter says that structural congruent nets have the same reductions.

Notice that the operational semantics presented so far is not intended to be the specification of how an actual implementation of the language should work. For example, repeatedly checking marked actions is useless and would degrade system performance. This problem can be avoided by exploiting an event-driven programming style: an event is associated to the acquisition of a given access right and marked actions are inserted in the associated event-listeners list.

We end this section by presenting two properties of the operational semantics, whose proofs can be found in Appendix A; as usual, we shall write $\rightarrowtail^*$ to denote the reflexive and transitive closure of $\rightarrowtail$. The first result relates the set L in a configuration $L \triangleright N$ to the names occurring in the net obtained after a reduction step. The second result states that, if we start with a net where pairwise distinct nodes have different addresses, such a property is preserved along reductions. Nets of this kind will be called *well-formed* and guarantee that each network node has a single access control policy, a very reasonable assumption in our setting. If not differently specified, in the sequel we shall only consider well-formed nets.

Proposition 4.2. *If* $L \triangleright N \rightarrowtail L' \triangleright N'$ *and* $\mathrm{fn}(N) \subseteq L$ *then* $\mathrm{fn}(N') \subseteq L'$.

Proposition 4.3. *If* N *is well-formed,* $\mathrm{fn}(N) \subseteq L$ *and* $L \triangleright N \rightarrowtail^* L' \triangleright N'$, *then* N' *is well-formed.*

[3]This permits splitting the parallel components running at a node and thus enables the application of the main reduction rules that, in fact, can only be used when there is a single process running at l. Moreover, by possibly using axiom (Abs) in Table 2, (Split) enables the use of axioms (Out), (Eval), (In) and (Read) also for execution of local operations. In conclusion, (Split) permits a compact and general formulation of the reduction rules without the need of explicitly considering all the parallel components running at a node and of having different rules for local and remote operations.

5. Correctness

We start by introducing the notion of *executable nets*; these are nets already containing all necessary marks, as if they have already passed a static checking phase.

Definition 5.1. *A net is* executable *if, for each node* $l ::^{\delta} C$*, it holds that* $\delta \vdash_l C \triangleright C$ *(that, for the sake of readability, will be written as* $\delta \vdash_l C$*).*

Notice that executable nets are admissible. Our main results will be stated in terms of executable nets; indeed, due to the dynamic acquisition of capabilities, well-formed nets that are statically deemed admissible can still give rise to run-time errors. However, by marking those actions that should be checked at run-time, admissible (and well-formed) nets can be transformed into executable nets that, instead, cannot give rise to run-time errors (see Theorem 5.7).

We first prove some results, i.e. weakening and substitutivity, which are standard for the theory of type systems.

Lemma 5.2 (Weakening). *If* $\Gamma \vdash_l^L C$ *then* $\Gamma[\Gamma'] \vdash_l^L C$.

Proof: The proof consists in mimicking for $\Gamma[\Gamma'] \vdash_l^L C$ the derivation of $\Gamma \vdash_l^L C$. The process actions enabled by Γ or those not enabled by Γ' will have the same judgments w.r.t. both Γ and $\Gamma[\Gamma']$; on the contrary, actions enabled by Γ' but not by Γ will be checked w.r.t. $\Gamma[\Gamma']$ using rules (T-SND)/(T-RCV) in place of rules (T-MSND)/(T-MRCV). □

Lemma 5.3 (Substitutivity). *If* $\Gamma \vdash_l^L C$ *then, for any substitution* σ, $\Gamma\sigma \vdash_l^{L'} C\sigma$, *where* $L' = L - dom(\sigma)$.

Proof: The proof is by induction on length of the inference of the judgment. The base cases (i.e., rules (T-NIL) and (T-DAT)) are trivial. Let us examine the case in which the last rule used is (T-RCV) (the cases for (T-REPL), (T-PAR), (T-SND), (T-NEW), (T-MSND) and (T-MRCV) are similar or easier). By hypothesis, we have that $C = a.Q$ and $\Gamma \vdash_l^L a.Q$, for some process Q and action a such that $ar(a) \in \{i, r\}$, $\{ar(a)\} \sqsubseteq_\Pi \Gamma(tgt(a))$ and $upd(\Gamma, arg(a)) \vdash_l^{L \cup \mathrm{BN}(arg(a))} Q$. Without loss of generality, we can assume that $dom(\sigma) \cap \mathrm{BN}(arg(a)) = \emptyset$ (otherwise, if this is not the case, we could rename the bound names); thus we have $(a.Q)\sigma = a\sigma.Q\sigma$. Now, by induction, we have that $(upd(\Gamma, arg(a)))\sigma \vdash_l^{L''} Q\sigma$, where $L'' = (L \cup \mathrm{BN}(arg(a))) - dom(\sigma) = (L - dom(\sigma)) \cup \mathrm{BN}(arg(a)) =$

$L' \cup \text{BN}(arg(a))$. Now, $upd(\Gamma\sigma, arg(a\sigma)) \vdash_l^{L''} Q\sigma$ (indeed, it is easy to prove that $(\Gamma_1[\Gamma_2])\sigma = (\Gamma_1\sigma)[\Gamma_2\sigma]$) and, by applying rule (T-RCV), we conclude that $\Gamma\sigma \vdash_l^{L'} a\sigma\,.\,Q\sigma$, i.e. $\Gamma\sigma \vdash_l^{L'} C\sigma$. □

Differently from [25] and from most type systems for calculi for network programming, the access control model we define in this paper permits a *local* formulation of correctness. 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 those nodes whose addresses are not in S. Now we prove that the property of a net of being executable is an invariant both of the structural congruence and of the reduction relation.

Lemma 5.4. *If $N|_S$ is executable and $N \equiv N'$ then $N'|_S$ is executable.*

Proof: By mutual induction on the length of the inferences for $N \equiv N'$ and $N' \equiv N$. The base case covers the axioms in Table 2. The cases of (COM) and (ASS) trivially follow by definition, the case for (ALPHA) follows from the fact that the static checking is not affected if we consistently rename bound names within a net, and the case for (ABS) is simple. Reflexivity is trivial, while the inductive steps, i.e. symmetry, transitivity and context closure, are easy. □

Theorem 5.5 (Subject Reduction). *If $N|_S$ is executable and $L \triangleright N \rightarrowtail L' \triangleright N'$ for $\text{FN}(N) \subseteq L$, then $N'|_{S'}$ is executable, where $S' = S \cup (L' \setminus L)$.*

Proof: The proof proceeds by induction on the length of the inference of $L \triangleright N \rightarrowtail L' \triangleright N'$.

Base Step: We reason by case analysis on the axioms (i.e. the first six rules) of Table 6.

(OUT). In this case, $S' = S$ because $L' = L$. Then, we have three possible subcases:

- *Both l and l' belong to S.* In this case, $N|_S = N$; since by hypothesis $N|_S$ is executable, we have that $\delta \vdash_l \mathbf{out}(t)@l'.P$ and $\delta' \vdash_{l'} C'$. By rule (T-SND), we have that $\delta \vdash_l P$. Moreover, by applying (T-PAR) to $\delta' \vdash_{l'} C'$ and to $\delta' \vdash_{l'} \langle t \rangle$ (axiom (T-DAT)), we get that $\delta' \vdash_{l'} C' \mid \langle t \rangle$. This suffices to conclude that $N'|_S (= N')$ is executable.
- *Neither l nor l' belong to S.* In this case, $N'|_S$ does not contain any node and, hence, is trivially executable.

- *One between l and l' belongs to S.* This case can be obtained by combining the previous two ones.

(Eval). This case is similar to the previous one. Just notice that, if $l' \in S$, we can prove $\delta' \vdash_{l'} P' \mid Q'$ by applying (T-par) to $\delta' \vdash_{l'} P'$, that holds by hypothesis, and to $\delta' \vdash_{l'} Q \triangleright Q'$, that is the premise of rule (Eval).

(In). In this case, the proof is non-trivial only if $l \in S$; so, let us assume that $l \in S$ and prove that $\delta[\delta''] \vdash_l P\sigma$. By hypothesis, we have that $\delta \vdash_l \mathbf{in}(T)@l'.P$, where rule (T-rcv) has been the last one applied to infer the judgment; hence, we also have that $upd(\delta, T) \vdash_l^{\mathrm{BN}(T)} P$. By definition, if $\{x_i : \pi_i\}_{i\in I}$ are the formal fields of T, we have that $upd(\delta, T) = \delta \uplus [x_i \mapsto \pi_i]_{i\in I}$. Moreover, by the premise of rule (In) and by Proposition 4.1, we have that $match_l^\delta(T, t) = \langle \delta'', \sigma \rangle$, where $\delta'' = [l_i \mapsto \pi_i]_{i\in I}$ and $\sigma = [{}^{l_i}/_{x_i}]_{i\in I}$. Now, $upd(\delta, T) = \delta \uplus [x_i \mapsto \pi_i]_{i\in I}$ implies that $upd(\delta, T)\sigma = \delta[l_i \mapsto \pi_i]_{i\in I} = \delta[\delta'']$. Thus, by applying Lemma 5.3 to $upd(\delta, T) \vdash_l^{\mathrm{BN}(T)} P$, we conclude that $\delta[\delta''] \vdash_l P\sigma$.

(Read). Similar to the previous case.

(New). In this case, $S' = S \cup \{l'\}$. If $l \notin S$, we trivially conclude, since $\delta' \vdash_{l'} \mathbf{nil}$. Otherwise, by hypothesis we have that $\delta \vdash_l \mathbf{newloc}(l' : \delta').P$, where rule (T-new) has been the last one applied to infer the judgment. Hence we also have that $\delta \uplus [l' \mapsto \delta(l)] \vdash_l^{\{l'\}} P$. The thesis follows by using Lemma 5.3 with substitution $\sigma = [{}^{l'}/_{l'}]$.

(Repl). If $l \notin S$, the case is trivial. Otherwise, $\delta \vdash_l * P$, that holds by hypothesis, implies that $\delta \vdash_l P$; the thesis follows by applying rule (T-par).

Inductive Step: We reason by case analysis on the last applied operational rule of Table 6.

(Mark). If $l \notin S$, then trivially $(l ::^\delta a.P \parallel l' ::^{\delta'} C')|_S$ is executable and the thesis follows by induction. Otherwise, we have that $\delta \vdash_l \underline{a}.P$; we explicitly consider only the case where a is a **in** or **read** (the case for **out** or **eval** is slightly easier). Due to the form of the process involved in the judgment, rule (T-mrcv) has been the last one applied to deduce the judgment; hence we also have that $upd(\delta, arg(a)) \vdash_l^{\mathrm{BN}(arg(a))} P$. By the premise of (Mark), we have that, when the reduction takes place, $\{ar(a)\} \sqsubseteq_\Pi \delta(tgt(a))$. Hence, by

applying (T-RCV), we can derive $\delta \vdash_l a.P$; this implies that $(l ::^{\delta} a.P \parallel l' ::^{\delta'} C')|_S$ is executable. The thesis now follows by induction.

(SPLIT). Like in the previous case, the proof is non-trivial only if $l \in S$. In this case, we have that $\delta \vdash_l C_1|C_2$. Due to the form of the process involved in the judgment, rule (T-PAR) has been the last one applied to deduce the judgment; hence we also have that $\delta \vdash_l C_1$ and $\delta \vdash_l C_2$. Thus, we have that $(l ::^{\delta} C_1 \parallel l ::^{\delta} C_2 \parallel N)|_S$ is executable and, by induction, we get that $(l ::^{\delta'} C_1' \parallel l ::^{\delta} C_2' \parallel N')|_{S'}$ is executable. It is easy to prove that $\delta \preceq \delta'$; thus, $\delta' = \delta[\delta'']$ for some δ''. Now, the thesis directly follows by using Lemma 5.2.

(PAR). The fact that $(N_1 \parallel N_2)|_S$ is executable (that holds by hypothesis) implies that both $N_1|_S$ and $N_2|_S$ are executable. By induction, $N_1'|_{S'}$ is executable. Moreover, we can prove that $N_2|_{S'} = N_2|_S$. Indeed, if $S' = S$ the claim is straightforward; if $S' \neq S$, we have that $S' = S \cup \{l'\}$, for some $l' \notin L$, and we can conclude by the fact that $\text{FN}(N_2) \subseteq L$. Thus, $(N_1' \parallel N_2)|_{S'}$ is executable.

(STRUCT). From the hypothesis, $N|_S$ is executable and $N \equiv N_1$; by Lemma 5.4, it follows that $N_1|_S$ is executable too. Now, by induction, we get that $N_2|_{S'}$ is executable. From this fact and from the hypothesis $N_2 \equiv N'$, again by Lemma 5.4, it follows that $N'|_{S'}$ is executable. □

Now, we introduce the notion of run-time error and prove safety, i.e. that executable nets do not give rise to run-time errors. *Run-time errors* are defined by the rules in Table 7 in terms of predicate $N \uparrow l$ that holds true when a process P located at a node in N with address l attempts to perform an action a that is not allowed by the policy δ of the node. The rules are straightforward. Notice that, since marked actions are checked at run-time, they cannot give rise to run-time errors. At most, when their execution is not permitted, the process that is trying to execute them is blocked, waiting for the acquisition of the corresponding capabilities by a parallel process running at the same node.

Theorem 5.6 (Safety). *If $N|_S$ is executable then $N \uparrow l$ for no $l \in S$.*

Proof: We prove the contrapositive, i.e. that if $N \uparrow l$ for some $l \in S$ then $N|_S$ is not executable. The proof is by induction on the length of the inference of $N \uparrow l$.

$$\frac{}{l ::^{\delta} a.P \uparrow l}\ (\text{ErrAct})\ \{ar(a)\} \not\subseteq_{\Pi} \delta(tgt(a)) \qquad \frac{N \uparrow l}{N \parallel N' \uparrow l}\ (\text{ErrPar}) \qquad \frac{N \equiv N' \qquad N' \uparrow l}{N \uparrow l}\ (\text{ErrStr})$$

Table 7: Run-time error

Base Step: In this case, the error is generated by using axiom (ErrAct). This means that N is a node of the form $l ::^{\delta} a.P$, for $l \in S$, and $\{ar(a)\} \not\subseteq_{\Pi} \delta(tgt(a))$. Therefore, node $l ::^{\delta} a.P$, and hence $N|_S$, is not executable otherwise action a would have been marked (see rules (T-snd) and (T-rcv) in Table 3, and the definition of function *mark*).

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

(ErrPar). By induction on the premise $N \uparrow l$ of the rule, we have that $N|_S$ is not executable. Hence, by definition, $(N \parallel N')|_S$ is not executable.

(ErrStr). By induction on the premise $N' \uparrow l$ of the rule, we have that $N'|_S$ is not executable. Then the thesis follows from the premise $N \equiv N'$ by using Lemma 5.4. □

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.

Theorem 5.7 (Correctness). *If $N|_S$ is executable and $L \triangleright N \rightarrowtail^{*} L' \triangleright N'$ for $\text{fn}(N) \subseteq L$, then for no $l \in S \cup (L' \setminus L)$ it holds that $N' \uparrow l$.*

Proof: The proof proceeds by induction on the length of $L \triangleright N \rightarrowtail^{*} L' \triangleright N'$. The base step is Theorem 5.6, while the inductive step follows from Theorems 5.5 and 5.6. □

To conclude, notice that a more traditional correctness result that involves the static checking of the whole net can be obtained simply by taking $S = \text{fn}(N)$. However, we insist that our formulation of correctness better fits the key features of open systems, where 'good' components usually run in hostile environments.

6. Example: Subscribing On-line Publications

In this section, we take up the publisher/subscriber scenario of Example 2.1 to show the μKLAIM's programming style and to illustrate a way to exploit its access control mechanism for enforcing access policies. For programming convenience, we shall assume integers and strings to be basic values of the language and omit trailing occurrences of process **nil**. Moreover, to suitably identify and refer to processes, we shall use notation $A \triangleq P$ to assign the name A to the process P.

Suppose that a user U wants to subscribe a 'license' to enable accessing on-line publications of a given publisher P. To model this scenario we use three localities, l_U, l_P and l_S, 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 the access right o) and credit card number; then, U waits for a tuple that will deliver it the access right r needed to access P's publications and proceeds with the rest of its activity. The behaviour described so far is implemented by the process

$$A_U \triangleq \mathbf{out}(\text{“}Subscr\text{”}, l_U : [l_P \mapsto \{o\}], CrCrd)@l_P.\mathbf{in}(\text{“}Acc\text{”}, !x : \{r\})@l_U.R$$

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

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

Concretely, the access right r will be delivered to U for a limited period of time (for example, annual subscriptions would obtain access rights valid for one year) or for a limited number of accesses. In Section 7.2 we shall present some simple ways to implement these features in our setting.

For processes A_U and A_P to behave in the expected way, the underlying net architecture, namely distribution of processes and access control policies, must be appropriately configured. A suitable net is:

$$\begin{aligned} & l_U ::^{[l_U \mapsto C,\, l_P \mapsto \{o\}]} \underline{A_U} \;\; \| \;\; l_P ::^{[l_P \mapsto C,\, l_S \mapsto \{o,i,r\}]} A_P \\ & \| \;\; l_S ::^{[\,]} \langle paper1 \rangle \mid \langle paper2 \rangle \mid \ldots \end{aligned} \tag{1}$$

where we have intentionally used $\underline{A_U}$ to emphasize the fact that the static checking might have marked some actions occurring in A_U, e.g. actions $\mathbf{read}(\ldots)@l_S$ in R.

Upon completion of the protocol, the net will be

$$l_U ::^{[l_U \mapsto C, l_P \mapsto \{o\}, l_S \mapsto \{r\}]} \underline{R} \;\|\; l_P ::^{[l_P \mapsto C, l_S \mapsto \{o,i,r\}, l_U \mapsto \{o\}]} A_P \;\|\; l_S ::^{[\,]} \langle paper1\rangle \mid \langle paper2\rangle \mid \ldots$$

Now consider the net

$$l_U ::^{\delta} Q \;\|\; l_P ::^{[l_P \mapsto C, l_S \mapsto \{o,i,r\}]} A_P \;\|\; l_S ::^{[\,]} \langle paper1\rangle \mid \langle paper2\rangle \mid \ldots \qquad (2)$$

If we can make assumptions on the policy δ, we can exploit our framework to state and guarantee some security properties.

- If $e \notin \delta(l_P)$ and $i \notin \delta(l_s)$, *availability* of P's papers is guaranteed in that only P can remove data from l_S, whatever process Q is. Indeed, Q could remove papers from l_S either by inputting them or by migrating at a node where this is allowed (viz., l_P). In the first case, $Q \triangleq \underline{\mathbf{in}}(paper)@l_S\,.Q'$, for some Q', where the action is marked because $i \notin \delta(l_S)$ and the net in (2) is executable. At run-time, the reference monitor will block Q for ever, since $i \notin \delta(l_S)$ and nobody in (2) is willing to pass the capability $l_S \mapsto \{i\}$ around. In the second case, $Q \triangleq \underline{\mathbf{eval}}(\mathbf{in}(paper)@l_S)@l_P\,.Q'$, for some Q', and we can reason in a similar way.

- Similarly, if $e \notin \delta(l_P)$ and $o \notin \delta(l_s)$, *integrity* of P's papers is ensured, in that only P can add data to l_S.

To conclude this section, we want to remark some features of this example that shed light on some peculiarities of our framework.

1. P's papers cannot be safely put in l_P's TS because otherwise the integrity of P's publications could be compromised by the execution at l_U of the legal process $\mathbf{out}(not\_a\_P\_paper)@l_P$. Indeed, our capability lists are not so refined to restrict the kind of tuples over which actions can operate: if $\mathbf{out}(\text{“}Subscr\text{”}, l_U : [l_P \mapsto \{o\}], CrCrd)@l_P$ has to be enabled, then also $\mathbf{out}(not\_a\_P\_paper)@l_P$ will be enabled: the executable net

   $$l_U ::^{\delta} \mathbf{out}(not\_a\_P\_paper)@l_P \;\|\; l_P ::^{[l_P \mapsto C]} \langle paper1\rangle \mid \langle paper2\rangle \mid \ldots$$

   evolves into

   $$l_U ::^{\delta} \mathbf{nil} \;\|\; l_P ::^{[l_P \mapsto C]} \langle paper1\rangle \mid \langle paper2\rangle \mid \ldots \mid \langle not\_a\_P\_paper\rangle$$

   where U has placed in l_P a paper not published by P. This problem can be avoided by exploiting the more refined policies we have introduced in [34].

2. Knowledge of address l_S is not enough for reading papers, the access right r is needed: access control in μKLAIM does not rely on name knowledge but on access control policies. Indeed, a process $Q \triangleq \underline{\mathbf{read}}(paper)@l_S\,.Q'$, for some Q', placed at l_U in (2) never reads papers, assuming that $r \notin \delta(l_S)$.
3. Once the access right r over l_S 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 license' valid for all processes running at l_U. This fact should not be considered as an access control breach: indeed, in order to enter l_U, a mobile process could be required to exhibit some credential (e.g. a password [44]), that however we do not model in our framework. Moreover, notice that this way of handling privileges is different from [25], where, by using the same protocol, U would have obtained a sort of 'individual license' for process R. In the next section we will present variations of our framework that permit delivering different capabilities to processes running at the same node.
4. The license delivered by P to U can be used only at l_U since the granting associated to l_S only delivers to l_U the access right r over l_S. Moreover, no intruder can remotely interfere with the protocol between the user and the publisher because the tuple $\langle$"Acc", $l_S : [l_U \mapsto \{r\}]\rangle$ located at l_U can only be retrieved by processes running at l_U (see rules (M_1) and (M_2) in Table 5). Indeed, if we add to (1) the node $l' ::^{\delta'} \mathbf{in}$("$Acc$", l_S)@l_U aiming at mounting a denial of service attack against l_U, such a node will not achieve its goal even if $i \in \delta'(l_U)$. A similar argument holds for the tuple $\langle$"$Subscr$", $\ldots\rangle$ inserted by A_U at l_P.

7. Variations on Capabilities Management

Up to now, capabilities are always acquired by the node hosting the process performing actions **in**/**read**, and not by the process itself. This may be adequate in some scenarios, e.g. when a department subscribes a 'site license' (i.e. valid for all its members), and unrealistic in others, e.g. when a mobile process has to buy a good on behalf of its owner. Moreover, capabilities can only increase; this is unsuitable to control wastable resources where one usually wants to count the number of times a given resource is used or to deliver accesses for a limited period of time.

In the next two subsections, we will show that our framework can be smoothly tailored for taking into account these different scenarios. For each variation, we shall first describe the scenario we want to model from an operational point of

view and present a concrete motivating example. Then, we shall discuss how the access control model can be tailored to preserve the results of Section 5.

Finally, in the last subsection we consider an orthogonal but realistic variation where some capabilities cannot be passed through. As it also happens in actual systems (see, e.g., [9, 27]), some capabilities can be passed while other, more critical, ones cannot.

7.1. Variations on Capabilities Acquisition

In this section, we show an adaption of our framework that allows processes to acquire capabilities for themselves. We start by presenting a scenario where all the dynamically acquired capabilities are assigned to single processes; then, we shall combine together the possibility of granting capabilities to processes and to nodes.

7.1.1. Acquisition by Processes

We start by modifying our framework to associate capabilities, in particular those dynamically acquired, to processes. To this aim, we annotate located processes with a capability list that specifies the capabilities they own. Thus, a process can also use its own private capabilities, in addition to the capabilities of the executing node that are shared by all co-located processes. Now, a μKLAIM node is of the form $l ::^{\delta} \mathcal{AC}$, where $\mathcal{AC}$ is an *annotated component* generated from the following syntactic productions

$$\mathcal{AC} \quad ::= \quad \langle t \rangle \quad \big| \quad \{\!\{ P \}\!\}_{\delta} \quad \big| \quad \mathcal{AC}_1 | \mathcal{AC}_2$$

Notice that only process components can be annotated.

The operational semantics is changed to manage the acquisition of capabilities that now increases process annotations while leaves policies of nodes unchanged. In the initial configuration, all processes could have assigned the same empty capability list or not, reflecting different capabilities for the processes. The adaptions are not surprising; they are in Table 8 and are reported in Appendix B. Notice that marked actions are now checked only against the capability list associated to the process performing them (see rule (MARK′)); indeed, the capability list of the node does never change and has already been used in the static checking phase.

Let us now briefly revise the subscription example. If in the initial configuration all processes have assigned the empty capability list, the evolution of the net (1) according to the modified semantics leads to

$$\begin{array}{l} l_U ::^{[l_U \mapsto C, l_P \mapsto \{o\}]} \{\!\{ \underline{R} \}\!\}_{[l_S \mapsto \{r\}]} \;\|\; l_P ::^{[l_P \mapsto C, l_S \mapsto \{o,i,r\}]} A_P \\ \|\; l_S ::^{[\,]} \langle paper1 \rangle \,|\, \langle paper2 \rangle \,|\, \ldots \end{array}$$

where now $\underline{R}$ is the only process having the capability to access the papers stored at l_S. Moreover, notice that the access right o over l_U delivered by A_U to A_P disappears upon completion of the parallel component running at l_P that handles A_U's request. Indeed, at the end of its task such a component becomes $\{\!\{ \mathbf{nil} \}\!\}_{[l_U \mapsto \{o\}]}$ and can be removed.

7.1.2. Acquisition by Nodes and Processes

In practice, a (mobile) process could acquire some capabilities 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. A simple way to model both cases is to use different acquisition actions depending on whether the acquisition should be made on behalf of the node or of the process. Hence, we could leave the operational semantics of actions **in**/**read** unchanged (i.e. as given in Section 4) apart for the replacement of processes with annotated processes, add actions $\mathbf{inpr}(T)@u$ and $\mathbf{readpr}(T)@u$ to the syntax, and model their operational semantics by using rules akin to (In′) and (Read′) in Table 8. In such a way, actions **in**/**read** would increase the capability list of the node where they are executed while actions **inpr**/**readpr** would increase the private capability list of the executing process.

Of course, to control the new actions, we also need to introduce the corresponding access rights and to extend the ordering relation over access rights. Furthermore, notice that, since node capability lists can dynamically change (like in the original semantics), in rule (Mark′) the hypothesis $\{ar(a)\} \sqsubseteq_\Pi \delta_1(l')$ must be replaced by $\{ar(a)\} \sqsubseteq_\Pi \delta_1(l') \cup \delta(l')$. Indeed, a marked action can be enabled both by the capabilities accumulated by the process and by the capabilities offered by the hosting node.

Correctness. We now sketch how the results of Section 5 can be adapted to the variation we have just presented (notice that the setting of Section 7.1.1 is clearly an instance of the model we develop here). The static checking mechanism needs smooth extensions: it should consider annotated processes and it should let rule (T-rcv) deal with actions **inpr**/**readpr** too. The first task can be carried out by adding the following inference rule

$$\text{(T-ann)} \qquad \frac{\Gamma[\delta] \ \vdash_l^L \ P \ \triangleright \ \underline{P}}{\Gamma \ \vdash_l^L \ \{\!\{ P \}\!\}_\delta \ \triangleright \ \{\!\{ \underline{P} \}\!\}_\delta}$$

A marked annotated component $\underline{\mathcal{AC}}$ is an annotated component that may contain annotated marked processes of the form $\{\!\{ \underline{P} \}\!\}_\delta$. Then, the notions of admissible

nets and executable nets are defined like before, but take into account annotated components.

Definition 7.1. *A net is* admissible *if, for each node* $l ::^{\delta} \mathcal{AC}$*, there exists a component* $\underline{\mathcal{AC}}$ *such that* $\delta \vdash_l \mathcal{AC} \triangleright \underline{\mathcal{AC}}$*. A net is* executable *if, for each node* $l ::^{\delta} \mathcal{AC}$*, it holds that* $\delta \vdash_l \mathcal{AC} \triangleright \mathcal{AC}$ *(abbreviated as* $\delta \vdash_l \mathcal{AC}$*).*

Finally, run-time errors are defined accordingly, by letting rule (ErrAct) become

$$\text{(ErrAct}') \qquad \frac{\{ar(a)\} \not\sqsubseteq_{\Pi} \delta(tgt(a)) \cup \delta'(tgt(a))}{l ::^{\delta} \{\!\{ a.P \}\!\}_{\delta'} \uparrow l}$$

Thus, correctness of the revised framework can be formulated and proved like in Theorem 5.7.

7.2. Managing Loss of Capabilities

In this subsection, we deal with some scenarios where capabilities can be lost. The three settings we shall present mainly differ in the formal definition of capabilities and in the way in which capabilities are lost. The main common feature is that the static checking mechanism is weakened since there are a lot of ingredients that can dynamically change. As it could be expected, more flexibility requires more run-time checks.

Since in this subsection we need to express capabilities removal, we introduce notation $\delta = \delta_1, \delta_2$ to denote that δ can be bipartitioned in δ_1 and δ_2. Formally, $\delta = \delta_1, \delta_2$ means that $\delta = \delta_1[\delta_2]$ and, for each $u \in dom(\delta_1) \cap dom(\delta_2)$, we have that $\delta_1(u) = \delta(u) - \delta_2(u)$ and $\delta_2(u) = \delta(u) - \delta_1(u)$. A similar notation is exploited also for grantings.

7.2.1. Consumption

If we interpret the 'acquisition of capabilities' as the 'purchase of services/goods', it is natural that a process will lose the acquired capability once it used the service. For example, by paying the price of a book a user purchases one copy of the book; if he wants another copy, he has to pay again. To enable multiple acquisitions and consumptions of capabilities, we should be able to count the number of capabilities that nodes/processes have over each resource (this is somehow similar to 'affine' types of [13]). To this aim, we modify our model by working with *multisets* of access rights, instead of sets; in particular, Π now denotes the set of the multisets built upon C (the set of access rights). All

the operations over and relations between sets used in this paper (i.e., union, subset inclusion, ...) must be considered as operations over and relations between multisets.

We start considering the case of dynamic acquisition and consumption of capabilities only by processes from Section 7.1.1. This means that node policies are statically known and left unchanged by the operational semantics. The operational rules are modified as reported in Table 9 (see Appendix B). The main change is that process capabilities must be deleted whenever used; this happens for actions **out** and **eval**, and when checking marked actions (see rules (OUT″), (EVAL″) and (MARK″)). Also pattern matching needs to be modified; now, when it is invoked by l on T and t, it returns a triple $\langle \delta'', \sigma, t' \rangle$. The difference is in the tuple t' obtained by removing from the grantings within t all the capabilities granted to l (i.e., the capabilities collected in δ''). This is necessary otherwise repeated accesses to a tuple via actions **read** would lead to a form of 'capability forging'. Indeed, each time a process at l reads t, the capabilities in δ'' would be delivered to the process. Since the **read** can be repeated several times (until $\langle t \rangle$ is available), it would be possible to acquire several times the capabilities δ''.

Taking up the example of Section 6, we can now program the acquisition (and the consumption) of a fixed number of access rights r over the on-line repository. The user explicitly requires a number k of access rights r and the publisher will charge on U's credit card the cost of k accesses to its publications. The processes implementing these behaviours are

$$
\begin{array}{rl}
A_U \triangleq & \mathbf{out}(\text{“}Subscr\text{”}, l_U : [l_P \mapsto \{o\}], CrCrd, k)@l_P. \\
& \mathbf{in}(\text{“}Acc\text{”}, !x : \{k \times r\})@l_U.R \\
A_P \triangleq & * \mathbf{in}(\text{“}Subscr\text{”}, !x : \{o\}, !y, !z)@l_P. \\
& \quad \textit{check credit card y of x and charge the cost for z accesses} \ . \\
& \quad \mathbf{out}(\text{“}Acc\text{”}, l_S : [x \mapsto \{z \times r\}])@x
\end{array}
$$

where $\{k \times r\}$ stands for the multiset with k occurrences of capability r.

Correctness. Differently from Section 7.1, process capabilities do not play any role in the static checking (thus, rule (T-ANN) is missing): indeed, since they can also decrease, it is statically impossible to rely on them to determine whether a given action will be legal at run-time or not. As an example, consider the net $l ::^{[\,]} \{\!\{ P|Q \}\!\}_{[l' \mapsto \{o\}]}$, where $P \triangleq \mathbf{out}(t)@l'$ and $Q \triangleq \mathbf{out}(t')@l'$. In this case, exactly one between P and Q will be able to perform action **out** while the other one will be blocked, depending on the execution order. However, it is impossible to statically

tell which one will evolve and which one will get stuck (and hence both of them have to be marked).

Furthermore, the static semantics now has to mark all the actions, except those directly enabled by the access policy of the node where the inference takes place. This is necessary to properly handle nodes like $l ::^{[l' \mapsto \{i\}]} \mathbf{in}(!u : \{o\})@l'.\mathbf{out}(\cdot)@l'.\mathbf{out}(\cdot)@u$, where action **in** should be the only unmarked one after static checking. Indeed, if we use the checking of Section 3.2, the second action **out** would not be marked. This could generate a run-time error if u is replaced by l' upon execution of the **in**: the acquired capability o, that enables execution of the second action **out**, would be consumed to perform the first action **out**.

Admissible and executable nets are formally defined like in Definition 7.1; run-time errors are defined like in Section 7.1.2, i.e. by exploiting rule (ErrAct′). Correctness can be still stated and proved similarly to Theorem 5.7.

A more general framework. Finally, let us now briefly consider the general setting where both processes and nodes can dynamically acquire and consume capabilities (see Section 7.1.2). This scenario is the most expensive because the static checking phase cannot be exploited at all and all actions must be checked at run-time. In fact, since also node capability lists can dynamically change, it is impossible to statically determine if a given action will have the necessary capabilities at run-time. Moreover, both the capability list associated to a process and the capability list of the node where the process is running can provide the process with the capability necessary to perform a given action. In this case, the capability can be removed from the capability list of the node or from the capability list of the process, and a strategy must be implemented. The operational rules can be easily modified to control capabilities and remove the used ones; to save space, we do not show the details.

7.2.2. Validity Duration

Another possible way of modeling capability lost is by introducing duration, as we already mentioned in the example of Section 6. Each capability can be assigned a validity duration by indexing it with a natural number or with the symbol ∞ representing the period of time during which the capability can be used: a capability is available until its validity has not been expired. Thus, capability lists (and grantings) map $\mathcal{N}$ to Π', where Π' is the powerset of $C \times (\mathbf{Nat} \cup \{\infty\})$ and it is ranged over by ρ. For example, $[l \mapsto \{i_{10}, o_5, e_\infty\}]$ expresses the fact that it is still possible to perform over l actions **in** for 10 time units, actions **out** for 5 time units and actions **eval** forever. Access rights like e_∞ will be called 'persistent' (notice

that all the access rights considered so far were indeed persistent).

The operational semantics of the basic framework needs to be modified to model time passing and the effect of time passing on validity durations. Because of the intrinsic asynchronous nature of our nets, we assume that time can pass differently in different parts of the net but, at each node, time passes uniformly for all the processes running there (this modeling is similar to web-π's one [42]). Moreover, we assume that time progresses in discrete time steps and label reductions with τ to indicate the passing of τ time units.

Technically, all the rules in Table 6, except (Par) and (Struct), represent computational steps and are assumed to be instantaneous; thus, the reductions occurring therein are labeled with '0'. The reductions contained in rules (Par) and (Struct) are instead labeled with a generic label τ because they can stand for computational steps or time steps. The following additional rule models time steps

$$\text{(Time)} \qquad l ::^{\delta} C \;\stackrel{\tau}{\succ\!\!\rightarrow}\; l ::^{(\delta)_{-\tau}} (C)_{-\tau}$$

Function $(\cdot)_{-\tau}$ is defined inductively as

$$\begin{array}{rcl}
(C_1 \mid C_2)_{-\tau} & \triangleq & (C_1)_{-\tau} \mid (C_2)_{-\tau} \\
(\langle t \rangle)_{-\tau} & \triangleq & \langle t' \rangle \quad \text{with } t' \text{ obtained from } t \text{ by replacing each } \mu \text{ with } (\mu)_{-\tau} \\
[\,]_{-\tau} & \triangleq & [\,] \\
([l \mapsto \rho])_{-\tau} & \triangleq & [l \mapsto \rho'] \\
 & & \text{where } \rho' \text{ is obtained from } \rho \text{ by:} \\
 & & \quad \bullet \text{ subtracting } \tau \text{ to all the durations, and} \\
 & & \quad \bullet \text{ deleting the access rights with a non-positive duration} \\
(\delta[\delta'])_{-\tau} & \triangleq & (\delta)_{-\tau}[(\delta')_{-\tau}] \\
(\mu[\mu'])_{-\tau} & \triangleq & (\mu)_{-\tau}[(\mu')_{-\tau}]
\end{array}$$

and it is the identity function in all the other cases. Thus, it can be easily seen that when τ_1 time units pass in l_1 and τ_2 time units pass in l_2, the net $l_1 ::^{\delta_1} C_1 \parallel l_2 ::^{\delta_2} C_2$ evolves as follows:

$$\begin{array}{rcl}
l_1 ::^{\delta_1} C_1 \parallel l_2 ::^{\delta_2} C_2 & \stackrel{\tau_1}{\succ\!\!\rightarrow} & l_1 ::^{(\delta_1)_{-\tau_1}} (C_1)_{-\tau_1} \parallel l_2 ::^{\delta_2} C_2 \\
 & \stackrel{\tau_2}{\succ\!\!\rightarrow} & l_1 ::^{(\delta_1)_{-\tau_1}} (C_1)_{-\tau_1} \parallel l_2 ::^{(\delta_2)_{-\tau_2}} (C_2)_{-\tau_2}
\end{array}$$

Correctness. We can statically control only the operations that are enabled by persistent access rights; all the other operations have to be marked, since it is not possible to exactly know when they will be performed. In particular, all the actions having a variable as target must be marked. Moreover, to avoid forging capability durations, we also need to ensure that a process delivers a capability with duration τ only if the capability is persistent or has a duration at least τ in the capability list of the node where the process runs.

These tasks can be achieved by defining an ordering on Π', written $\sqsubseteq_{\Pi'}$, as follows

$$\frac{\tau' \leq \tau}{\{c_{\tau'}\} \sqsubseteq_{\Pi'} \{c_\tau\}} \qquad \frac{\rho_1 \subseteq \rho_2}{\rho_1 \sqsubseteq_{\Pi'} \rho_2} \qquad \frac{\rho_1 \sqsubseteq_{\Pi'} \rho_1' \qquad \rho_2 \sqsubseteq_{\Pi'} \rho_2'}{(\rho_1 \cup \rho_2) \sqsubseteq_{\Pi'} (\rho_1' \cup \rho_2')}$$

Clearly $\preceq$, $[\![\cdot]\!]_\delta$, $match_l^\delta(\cdot,\cdot)$ and $mark_\Gamma^L(\cdot)$ now exploit this ordering. In particular, this fact implies that, since $ar(a)$ returns an access right that is *not* annotated, action a is marked whenever a corresponding persistent capability is missing in the current checking context. On the other hand, rule (Mark) still invokes $\sqsubseteq_\Pi$, that can be straightforwardly extended to annotated access rights by ignoring durations.

The notions of admissible nets and executable nets are still defined like in Definitions 3.6 and 5.1. Correctness is then formulated and proved like in Theorem 5.7: it relies on the run-time errors defined in Table 7, that are still defined in terms of $\not\sqsubseteq_\Pi$ (properly extended to ignore validity durations). The only difference is that, in stating and proving subject reduction (Theorem 5.5), we also need to consider time passing, i.e. reductions of the form $\overset{\tau}{\rightarrowtail}$.

7.2.3. Revocation

We shall now touch upon a scenario where capabilities can be revoked, i.e. a node can delete capabilities of other nodes. To rule out obvious nasty attacks, we allow l to remove a capability list δ from l' only if l has previously passed a list greater than δ to l' (notice that this complies with standard trends in discretionary access control models). In doing so, we have also to take into account the fact that several nodes could have passed δ to l'.

We let $\mathcal{S}$ to be the set of the finite subsets of $\mathcal{N}$ and we let $s, s', \ldots$ to range over $\mathcal{S}$. We now annotate access rights with the identity of the deliverers, thus obtaining the set of *annotated access rights* Π', ranged over by ρ. Formally, Π' contains the subsets of $C \times \mathcal{S}$ such that, if $(c, s_1) \in \rho$ and $(c, s_2) \in \rho$, then $s_1 = s_2$. Statically assigned access rights take the form $(c, \emptyset)$. We let the preorder $\sqsubseteq_{\Pi'}$ on annotated access rights to be defined by the following rules:

$$\frac{s_1 \subseteq s_2 \ \vee \ s_2 = \emptyset}{\{(c, s_1)\} \sqsubseteq_{\Pi'} \{(c, s_2)\}} \qquad \frac{\rho_1 \subseteq \rho_2}{\rho_1 \sqsubseteq_{\Pi'} \rho_2} \qquad \frac{\rho_1 \sqsubseteq_{\Pi'} \rho_1' \quad \rho_2 \sqsubseteq_{\Pi'} \rho_2'}{(\rho_1 \cup \rho_2) \sqsubseteq_{\Pi'} (\rho_1' \cup \rho_2')}$$

Grantings are left unchanged, i.e. they are finite partial functions from $\mathcal{N}$ to Π, while capability lists now use annotated access rights. We use γ to range over these annotated capability lists that, formally, are finite partial functions mapping $\mathcal{N}$ to Π'. For example, the capability list $[l \mapsto \{(i, \{l_1\}), (o, \{l_2, l_3\})\}]$ used as access control policy of node l' enables actions **in**/**out** from l' over l, and records that the capability i has been delivered by l_1 while the capability o has been delivered by both l_2 and l_3. The ordering relation between annotated capability lists, $\preceq'$, is defined like $\preceq$ but relies on $\sqsubseteq_{\Pi'}$ instead of $\sqsubseteq_{\Pi}$. If γ_1 and γ_2 are annotated capability lists, the extension $\gamma_1[\gamma_2]$ is the annotated capability list γ' such that

$$\gamma'(u) \triangleq \begin{cases} \gamma_1(u) & \text{if } u \in dom(\gamma_1) - dom(\gamma_2) \\ \gamma_2(u) & \text{if } u \in dom(\gamma_2) - dom(\gamma_1) \\ \gamma_1(u) + \gamma_2(u) & \text{if } u \in dom(\gamma_1) \cap dom(\gamma_2) \end{cases}$$

where $\rho_1 + \rho_2$ is inductively defined as follows

$$\begin{array}{rcl} \emptyset + \rho & \triangleq & \rho \\ \{(c, s)\} + \rho & \triangleq & \begin{cases} \{(c, s \uplus s')\} \cup \rho' & \text{if } (c, s') \in \rho \text{ and } \rho' = \rho - \{(c, s')\} \\ \{(c, s)\} \cup \rho & \text{if } (c, \_) \notin \rho \end{cases} \\ (\{(c, s)\} \cup \rho) + \rho' & \triangleq & \{(c, s)\} + (\rho + \rho') \end{array}$$

We let $s_1 \uplus s_2$ be $s_1 \cup s_2$ if both $s_i \neq \emptyset$, and $\emptyset$ otherwise. Underlying the definition of $\uplus$ there is the assumption that, if a capability has been statically assigned to a given node (and hence one of the s_i is the empty set), then no other node will ever be allowed to revoke it; a similar motivation inspired us the definition of $\sqsubseteq_{\Pi'}$.

To enable capability revocations, we add action **revoke**$(\delta)@u$ to the syntax of μKLAIM actions. The operational rules are in Table 10 in Appendix B. Mainly, we have to deal with revocations: to this aim, we have to verify that the revoked capabilities, δ, are present in the capability list γ' of l' and that l was one of the grantors of δ in γ'. To enforce this requirement we 'sign' a tuple with the identity of the producer; in this way, when capabilities contained in the tuple are acquired, the identity of the granter is properly recorded to enable future revocations. This can be obtained by letting located tuples take the form $\langle t \rangle^l$, where l is the producer of the tuple. Then, when a policy is updated after a **read**/**in** by exploiting capabilities passed by a node l'' (see rules (IN$'''$) and (READ$'''$)), the received capabilities are annotated with l''.

We now show two possible uses of **revoke** in the example of Section 6. The first use consists in an alternative way of implementing the subscription for a fixed period of time d. Indeed, if we do not introduce validity durations as previously shown, we can let P to manage timing information: once U's capability r has expired, P can revoke it. A simplified process A_P implementing this behaviour is

$$\begin{array}{rl} A_P \triangleq & *\,\mathbf{in}(\text{“}Subscr\text{”}, !x : \{o\}, !y, !d)@l_P. \\ & \quad \textit{check c.c. } y \textit{ of } x \textit{ and require the payment for duration } d. \\ & \quad \mathbf{out}(\text{“}Acc\text{”}, l_S : [x \mapsto \{r\}])@x.\mathbf{out}(x, Today() + d)@l'_P.B \\ B \triangleq & *\,\mathbf{in}(x, !s)@l'_P. \\ & \quad \mathbf{out}(\text{“}check\text{”}, x, Today(), Today() \leq s)@l'_P. \\ & \quad \big(\mathbf{in}(\text{“}check\text{”}, x, Today(), \mathbf{false})@l'_P.\mathbf{revoke}([l_S \mapsto \{r\}])@x \\ & \quad\ \mid \mathbf{in}(\text{“}check\text{”}, x, Today(), \mathbf{true})@l'_P.\mathbf{out}(x, s)@l'_P \big) \end{array}$$

where l'_P is a reserved locality where P stores timing information (we have silently used basic values representing dates and booleans, together with some obvious operations over them). Intuitively, process A_P handles timing expirations by recording in l'_P the expiration date of U's subscription, given by $Today() + d$. Then, process B repeatedly verifies the validity of the subscription by checking whether the current date, given by function $Today()$, is antecedent to the expiration date of U's subscription. When expired, the capability enabling the access to P's papers is revoked.

Another possible use of **revoke** in our example consists in revoking the access capability to a misbehaved user, e.g. a user that sold the acquired capability r to a third part at a lower price. Notice, however, that evidence of U's crime cannot be implemented in our calculus; also in practice there would be an external authority entitled to discover the crime and inform the publisher.

Correctness. We now adapt the static checking mechanism of Section 3.2 to the new scenario. First, notice that we do not need a specific capability to enable **revoke**: the operation is enabled only if l has previously delivered δ to l', and this is checked at run-time. Hence, the static checking mechanism is modified by using $\preceq'$ in rule (T-NEW) and by adding the following rule

$$(\text{T-REV}) \qquad \frac{\Gamma \vdash^L_l P \triangleright \underline{P}}{\Gamma \vdash^L_l \mathbf{revoke}(\delta)@l'.P \triangleright \mathbf{revoke}(\delta)@l'.\underline{P}}$$

Like for the previous variations, the checking can only rely on statically assigned capabilities; indeed, annotated capabilities can be revoked in unpredictable ways.

Again, this forces us to also mark all those actions whose target is a variable because we cannot know if the action will be enabled by a revocable capability or not.

Admissible and executable nets are defined by relying only on statically assigned rights. To this aim, we use function $pol(\gamma)$, that yields a simple capability list δ by deleting from γ all capability annotations, and $static(\gamma)$, that is the annotated capability list obtained from γ by removing all the capabilities that have not been statically assigned.

Definition 7.2. *A net is* admissible *if, for each node $l ::^{\gamma} C$, there exists a component $\underline{C}$ such that*

$$pol(static(\gamma)) \vdash_l C \triangleright \underline{C}$$

A net is executable *if, for each node $l ::^{\gamma} C$, it holds that*

$$pol(static(\gamma)) \vdash_l C \triangleright C$$

The definition of run-time errors now relies on the following variant of rule (ErrAct)

$$\frac{\{ar(a)\} \not\sqsubseteq_{\Pi} pol(\gamma)(tgt(a))}{l ::^{\gamma} a.P \uparrow l}$$

and correctness of the revised framework can be formulated and proved like in Theorem 5.7.

Notice that, here and in the other variations on capability loosing, the correctness theorems can be essentially proved like in Section 5. This is due to the fact that, for the static checking mechanism, we only consider the capabilities that are always available, i.e. those capabilities that cannot be consumed, that never expire and that cannot be revoked. The marking mechanism, that does never give rise to run-time errors, is exploited whenever capabilities that can become unavailable are required.

Possible extensions. The scenario we have just presented is perhaps the simplest way to model revocation of capabilities. We conclude by touching upon more elaborated scenarios.

- According to rule (Revoke), the process **revoke**$(\delta)@l'.P$ is stuck if only a list of capabilities less than δ is present in γ'. If we want to avoid this, we can adapt the operational rule for **revoke** to remove from γ' the greatest sublist of δ delivered by l.

- The proposed formulation rules out direct attacks aimed at revoking as many capabilities as possible to reduce the functionality of a system. These attacks can be mounted by executing actions **revoke**$(\delta)@l'$ by a process running at l, where l did not delivered δ to l'. However, one can easily imagine a scenario in which l spawns such a malicious process over an l'' that delivered δ to l'. A simple way to avoid this is to define two checking systems: the first one is $\vdash_l$, the other one, denoted by $\Vdash_l$, is defined as the first one but without rule (T-REV). We still use $\vdash_l$ in the definitions of admissible nets and of executable nets, while we use $\Vdash_l$ in rule (EVAL): in this way we block incoming agents containing actions **revoke**. This solution can however be over-restricting: a better (but more complex) solution is to define $\Vdash_l$ in such a way that **revoke**$(\delta)@l'$ is deemed legal only if it is syntactically preceded by an action **out** delivering l' some capability list greater than δ.

- The last scenario we consider is when l_1 delivers δ to l_2 and then l_2 delivers δ to l. Should it be legal for l_1 to perform an action **revoke** over l? In the current framework it is not. However, we could model this scenario by annotating access rights with subsets of $\mathcal{S}$; each such subset would represent an unordered path leading to the acquisition of the capability. E.g., if c is annotated with the set $\{\{l_1, l_2\}, \{l'_1, l'_2, l'_3\}\}$ in the annotated capability list of l, then c has been delivered to l through l_1 and l_2 and, independently, through l'_1, l'_2 and l'_3. Clearly, the semantics has to be modified to enable all the l_is and l'_js to perform actions **revoke** over l.

7.3. Managing Distribution of Capabilities

We conclude by dealing with an orthogonal feature of capability-based access control systems, namely the ability of controlling capability distribution. Usually, in discretionary access control models or in delegation-based trust models (see, e.g., KeyNote [9] and SPKI [27]), some capabilities can be granted while some other ones cannot. Moreover, the 'grantable' capabilities can be passed with an explicit indication that they cannot be further granted. We show how distribution of access rights can be integrated in our basic model (Sections 3 and 4); integration in the more sophisticated scenarios presented in this Section can be carried out similarly.

We start by defining the set of *labeled access rights* to be $C \times \{\circ, \bullet\}$, ranged over by λ; for notational convenience, we put the labels $\circ$ and $\bullet$ as superscripts to access rights. Sets of labeled access rights are grouped in Ξ that is ranged over by ξ. Capabilities, capability lists and grantings are now defined w.r.t. Ξ instead of

Π. Intuitively, an access right labeled with '$\circ$' is grantable, while an access right labeled with '$\bullet$' is not. Thus, the capability $l \mapsto \{i^\circ, o^\bullet\}$ denotes the possibility of further granting the access right i but not the access right o.

Resting on the idea that a grantable access right might also not be granted, while the converse must be avoided, we now define the ordering on sets of labeled access rights, $\sqsubseteq_\Xi$, as the least transitive relation closed under the following rules:

$$\{c^\circ\} \sqsubseteq_\Xi \{c^\circ\} \qquad \{c^\bullet\} \sqsubseteq_\Xi \{c^\circ\} \qquad \xi \sqsubseteq_\Xi \xi \cup \xi' \qquad \frac{\xi_1 \sqsubseteq_\Xi \xi_1' \qquad \xi_2 \sqsubseteq_\Xi \xi_2'}{\xi_1 \cup \xi_2 \sqsubseteq_\Xi \xi_1' \cup \xi_2'}$$

Notice that $\sqsubseteq_\Xi$ is *not* reflexive, because $\{c^\bullet\} \sqsubseteq_\Xi \{c^\bullet\}$ does not hold. This is due to the fact that $\sqsubseteq_\Xi$ is used to govern capability passing and that a non-grantable access right cannot be passed. Indeed, the checking of grantings in Table 4 and the ordering on capability lists given in Definition 3.3 now exploit $\sqsubseteq_\Xi$ instead of $\sqsubseteq_\Pi$. However, the run-time check of rule (Mark) and the rules for run-time error in Table 7 still rely on $\sqsubseteq_\Pi$. Also the pattern matching in Table 5 relies on $\sqsubseteq_\Xi$; in particular, rule (M_2) now becomes

$$\frac{\xi \sqsubseteq_\Xi \delta(l') \cup \mu(l) \qquad ar(\xi) = \pi}{match_l^\delta(!\, x : \pi, l' : \mu) = \langle [l' \mapsto \xi], [^{l'}/_x] \rangle}$$

where, with abuse of notation, we use function $ar(\cdot)$ to also remove all labels from a set of labelled access rights.

Finally, we are left with the definition of extension of capability lists. To this aim, we let $sup(\cdot,\cdot)$ be the least reflexive and symmetric function over labeled access rights such that $sup(c^\circ, c^\bullet) = c^\circ$. Function $sup(\cdot,\cdot)$ is then extended to sets of labeled access rights as follows

$$sup(\xi_1, \xi_2) \triangleq \begin{cases} \{\lambda\} \cup sup(\xi_1', \xi_2) & \text{if } \lambda \in \xi_1 \text{ and } \xi_1' = \xi_1 - \{\lambda\} \\ & \text{and } \forall \lambda' \in \xi_2 \,.\, ar(\lambda) \neq ar(\lambda') \\ sup(\lambda_1, \lambda_2) \cup sup(\xi_1', \xi_2') & \text{if } \lambda_1 \in \xi_1 \text{ and } \xi_1' = \xi_1 - \{\lambda_1\} \\ & \text{and } \lambda_2 \in \xi_2 \text{ and } \xi_2' = \xi_2 - \{\lambda_2\} \\ & \text{and } ar(\lambda_1) = ar(\lambda_2) \\ \xi_2 & \text{if } \xi_1 = \emptyset \end{cases}$$

Now, we let $\delta_1[\delta_2]$ be the capability list δ such that

$$\delta(u) \triangleq \begin{cases} \delta_1(u) & \text{if } u \in dom(\delta_1) - dom(\delta_2) \\ \delta_2(u) & \text{if } u \in dom(\delta_2) - dom(\delta_1) \\ sup(\delta_1(u), \delta_2(u)) & \text{if } u \in dom(\delta_1) \cap dom(\delta_2) \end{cases}$$

The rationale underlying this definition is that a non-grantable access right can be upgraded because of extension with the corresponding grantable access right.

To test the impact that this variation has on the expressiveness of our model, we reconsider the example of Section 6. In that scenario, every user could pass through the capability $l_S \mapsto \{r\}$ received by the publisher, thus acting as a tricky contender of P. By exploiting labeled access rights, we can model P's behaviour in a safer way by letting

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

Now, consider the net

$$N \triangleq l_U ::^{\delta} Q \;\; \| \;\; l_P ::^{[l_P \mapsto C \times \{\bullet\}, l_S \mapsto \{o^\bullet, i^\bullet, r^\circ\}]} A_P \;\; \| \;\; l_S ::^{[\,]} \langle paper1 \rangle \,|\, \langle paper2 \rangle \,|\, \ldots$$

If we assume that

$$\{e^\circ, e^\bullet\} \cap \delta(l_P) = \emptyset \;\text{ and }\; \{e^\circ, e^\bullet, i^\circ, i^\bullet, r^\circ, r^\bullet\} \cap \delta(l_s) = \emptyset \tag{3}$$

then we can prove that, whatever process Q is, data at l_S can only be accessed by l_U in **read**-mode and after the payment has been checked. Thanks to non-grantable access rights, this property also holds in $N \,\|\, M$, for every M whose node policies respect the assumptions made in (3) for the policy δ of l_U.

The correctness of the resulting model can be easily established by following the steps presented in Section 5. To save space, we omit the details.

8. Related Work

Protection mechanisms for shared data-spaces coordination languages. Several protection mechanisms have been proposed for shared data-space coordination languages that, like μKLAIM, are based on LINDA. Here, we describe the approaches closer to ours and refer the interested reader to [28] for a survey of other approaches.

Some works use cryptographic mechanisms for protecting data items, tuples and tuple spaces. For example, SecSpaces [36] associates a label to any protected tuple specifying the key needed to unlock the tuple and the modality (i.e., via 'read' or 'in' operations) in which it can be accessed. Labels can be inserted within data fields, thus privileges can be dynamically acquired trough communication. In [38], Lime (a framework for programming ad hoc networks via mobile

processes transiently sharing tuple spaces) is enriched with a password-based access control mechanism that permits the access to tuples and tuple spaces only to the processes that know the appropriate passwords. The initial password distribution is possibly accomplished outside of the application itself, while password exchanges are managed by the application. These programming choices are very similar to the ones for μKLAIM the we adopted in this paper.

Cryptographic keys, labels and passwords are similar approaches to protect single tuples, that overcome the impossibility for capabilities to refer anonymous objects. An alternative approach is put forward in [60] with the introduction of Lindacap, a LINDA-like capability-based system with *multicapabilities*. Multicapabilities provide a partitioning of a tuple-space and enable certain operations to be performed on tuples of a specific group, but not on those of another group, even though both groups have the same template. A multicapability may be copied to be passed to other processes; moreover, some operations on capabilities are introduced (e.g., set-like union, intersection and difference). Similar finer-grained capabilities for μKLAIM have been introduced in [34], where capabilities also specify a template for tuples, i.e. the argument of an operation in addition to its type. The partitioning of the tuple spaces provided by multicapabilities can somehow be mimicked by exploiting μKLAIM dynamically created tuple spaces, although μKLAIM lacks the combination calculus of multicapabilities. Moreover, Lindacap only uses dynamic checking whereas μKLAIM relies on both static and dynamic checking.

Distributed process calculi with protection mechanisms. A number of process calculi with distribution and mobility have been equipped in the last decade with protection mechanisms based on, e.g., type systems [25, 13, 41, 11, 15], control/data flow analysis [47, 48, 26, 40] and flow logic [39].

The approach closest to ours is the one based on type systems. However, among the large amount of work on type systems for resource protection in calculi with process distribution and mobility, only [51, 22, 12] handle dynamic modification of security policies. In [51] dynamic modifications of local knowledge of nodes are allowed, but must always respect a *global* policy for the net. Thus, the global policy is fixed at the beginning and does never change. The work in [22] somehow adapts our approach to the Ambient Calculus, where local policies are modified as an effect of ambient mobility. However, the way in which an ambient movement modifies a local policy is hardcoded within the moving ambient; this fact reduces the flexibility of the approach. In [12] the authors develop a secure implementation of a typed π–calculus, in which capability-based types

are employed to regulate the access to communication channels and dynamically exchange access rights between processes. High-level π–calculus processes are translated into low-level principals of a cryptographic process calculus which is a variant of the 'applied' π–calculus [2]. The high-level type capabilities are implemented as term capabilities protected by encryption keys only known to the intended receivers. As such, the implementation is effective even when the compiled, low-level principals are deployed in open contexts for which no assumption on trust and behavior may be made. This approach is refined even further in [4, 30] by implementing high-level functionalities directly using computational cryptography.

Other related approaches. Software capabilities have also been used to build a protection scheme for the Java environment [37]. As in our framework, access rights can be dynamically exchanged via communications by mutually suspicious processes. However, in our model capabilities are made available at the programming level (e.g. through grantings that are used explicitly for exchanging access rights), while in *loc. cit.* access control is handled as a non-functional aspect defined at the level of application interface and is completely separated from the functional code of applications.

In [3], the access rights of a piece of code are determined by examining the attributes (e.g. accessed data, site of origin, and so on) of the pieces of code that have run before and any explicit requests to augment rights. In other words, the access rights of a process depend on the history of its execution and of control transfers among processes. Instead, we consider a very simple and abstract process language and most of the ideas put forward by [3] do not apply. Some features, however, can be easily integrated in our setting. For example, as we show in [34], we can set node policies to grant capabilities to incoming processes according to the nodes spawning them, thus taking into account their execution history (i.e. the nodes they have already passed through).

In the last few years, several security frameworks for open systems appeared in the literature [59, 16, 17, 55, 6] that, similarly to ours, combine static and dynamic checks for efficiency and flexibility matters. We linger on the most related approaches. In [59], run-time principals are introduced for specifying information-flow security policies also in terms of information available at run-time (e.g., which principals will interact with the system). Dynamic checks are used to inspect run-time principals to determine policy information not available at compile time. Similarly, in our setting, processes can exploit capabilities dynamically acquired by weakening the static checking and by delaying some checks at

run-time. In [17], secrecy properties are guaranteed for a variant of the π-calculus with filesystem constructs. The calculus supports both access control checks and a form of static scoping that limits the knowledge of terms, including file names and contents, to groups of clients. As in our approach, while the typing is static, it applies to a program subject to dynamic access-control checks. In [55] the static and the dynamic approach to information flow are compared, to better understand their strengths and weaknesses. In general, since concrete values are known at run-time, run-time analyses can achieve greater precision and are more suitable to support security policies that are defined dynamically. On the contrary, static analyses must reject entire programs as insecure, where a run-time system needs only reject insecure executions of a program, but are more efficient. Our proposal aims at taking advantage of both approaches by merging them.

Another related research line concerns the definition of languages for dynamically evolving security policies [18, 57, 7]. In particular, [57] studies dynamic policies as channels that carry sensible information and develops a static type system that ensures a form of non-interference. A similar research line is followed in [7]. In [18], a framework is presented where, when analyzing a system statically, there may be available only partial knowledge of the structure of security policies. Similarly to ours, the framework permits static reasoning even when only partial knowledge of the run-time security policy structure is available.

To conclude, it has to be said that we have used a very simple policy language. A challenging issue for future research is the extension of our framework for dealing with policies written in a more complex policy language as, e.g., one of the languages surveyed in [1].

9. Conclusions

We presented μKLAIM, a foundational calculus for network aware programming, and its capability-based access control model. The latter permits controlling process mobility and enforcing protection of resources against misuse; moreover, it enables access control management by governing the use of resources and selectively distributing capabilities to processes. According to the terminology used in [53], our framework exploits a combination of static and dynamic checking, and of in-lined reference monitoring implemented by marking those process actions that need run-time verification. We have also presented some variations of the basic framework that enable processes to acquire capabilities for themselves, take into account capabilities loss and permit to constraint capabilities distribution. With respect to more traditional approaches exploiting capability-based access control,

preliminary static checks are introduced and performed everywhere possible to increase efficiency. However, due to process migration and dynamic modifications of access control policies, run-time checks are still largely used.

Our model is largely independent from the underlying language and from the definition of access rights. More specifically, it is possible to define a model similar to the one for μKLAIM we have presented in this paper, whenever we have: (*i*) a language with a set of process operations and a set of corresponding access rights, (*ii*) an ordering relation over sets of access rights, and (*iii*) some linguistic primitives for exchanging capabilities. For example, it is conceptually easy to adapt the current framework to the access rights used in [34], where finer-grained capabilities are exploited (by taking into account also the argument of an operation) and where a host can assign different privileges to processes coming from different nodes. Clearly, these are orthogonal features that can be integrated in our framework; however, to keep the notations in this paper simple, we have preferred to omit them.

A. Proofs of Technical Results

In this section we shall prove some technical results stated in the paper, namely Propositions 3.5, 4.2 and 4.3.

Proposition 3.5. *For any* Γ*,* l*,* L*,* C *and* C' *it is decidable to determine whether the judgment* $\Gamma \vdash_l^L C \triangleright C'$ *holds true or not.*

Proof: We firstly introduce the function $\#(C)$ that gives an upper bound to the number of checking rules that must be applied to establish the validity of a judgment $\Gamma \vdash_l^L C \triangleright C'$.

$$\#(C) \triangleq \begin{cases} 1 & \text{if } C = \langle t \rangle \text{ or } C = \mathbf{nil} \\ 1 + \#(P) & \text{if } C = a.P \text{ or } C = \underline{a}.P \text{ or } C = *P \\ 1 + \#(P_1) + \#(P_2) & \text{if } C = P_1 | P_2 \end{cases}$$

Notice that $\#(C)$ is always linear in the number of operators occurring in C, hence it is finite and does not depend on Γ or L. We then prove the following lemma that trivially implies the thesis.

> **Lemma A.1.** *For any* Γ*,* l*,* L*,* C *and* C' *the validity of judgment* $\Gamma \vdash_l^L C \triangleright C'$ *can be established in at most* $\#(C)$ *inference steps. In particular, exactly* $\#(C)$ *rules are needed to validate the judgment, while a smaller number is needed to disprove it.*

Proof: The proof is by induction on $\#(C)$. The key observation is that the inference of the judgment $\Gamma \vdash_l^L C \triangleright C'$ is driven by the syntax of C itself; hence, at any step at most one rule can be applied.

Base case: $\#(C) = 1$**.** We reason on the syntax of C.

- $C = \langle t \rangle$. In this case, the only applicable static checking rule is (T-DAT) that permits deducing $\Gamma \vdash_l \langle t \rangle \triangleright \langle t \rangle$. Thus, the judgement $\Gamma \vdash_l C \triangleright C'$ is valid if, and only if, $C' = \langle t \rangle$ and this can be established in one step.
- $C =$ **nil**. The proof proceeds similarly, once we replace $\langle t \rangle$ with **nil** and (T-DAT) with (T-NIL).

Inductive case: $\#(C) > 1$**.** We reason on the syntax of C.

- $C = a.P$. We further distinguish the case where a is an action **newloc** from the case where a is another action.
  - $a =$ **newloc**$(l' : \delta)$. Due to the syntax of C, the only static checking rule that could be applied is (T-NEW). For (T-NEW) to be applicable it must hold that $\delta \preceq \Gamma \uplus [l' \mapsto \Gamma(l)]$; otherwise, $\Gamma \vdash_l^L C \triangleright C'$ would not hold. Moreover, it must hold that $C' =$ **newloc**$(l' : \delta).P'$ for some P' such that $\Gamma[l' \mapsto \Gamma(l)] \vdash_l^{L \cup \{l'\}} P \triangleright P'$. Since, by definition, $\#(C) = 1 + \#(P)$, by induction we conclude that:
    - if such P' does not exist, then this can be determined by using less than $\#(P)$ steps, and hence we can confute $\Gamma \vdash_l^L C \triangleright C'$ by using less than $\#(C)$ steps;
    - otherwise, $\#(P)$ steps are needed for P and one step is needed to apply (T-NEW). Thus, we can validate $\Gamma \vdash_l^L C \triangleright C'$ by using $\#(C)$ steps.
  - $a \neq$ **newloc**$(\cdots)$. We can only apply rules (T-SND) or (T-RCV). They both require $C' = mark_\Gamma^L(a).P'$ for some P' such that $\Gamma \vdash_l^L P \triangleright P'$ or $upd(\Gamma, arg(a)) \vdash_l^{L \cup \mathbf{BN}(arg(a))} P \triangleright P'$, respectively. The thesis then follows by induction.
- $C = \underline{a}.P$. This case proceeds like the case for $a \neq$ **newloc**$(\cdots)$ but uses (T-MSND)/(T-MRCV) in place of (T-SND)/(T-RCV).

$C = C_1 | C_2$. The only checking rule that can be used in this case is (T-PAR). To this aim, C' must be of the form $C'_1 | C'_2$ for some C'_1 and C'_2 such that $\Gamma \vdash^L_l C_i \triangleright C'_i$ for $i = 1, 2$. By using a straightforward induction on the latter judgments, the thesis follows.

$C = * P$. The only checking rule that can be used in this case is (T-REPL). To this aim, C' must be of the form $* P'$ for some P' such that $\Gamma \vdash_l P \triangleright P'$. The thesis follows by induction. □

Proposition 4.2. *If* $L \triangleright N \rightarrowtail L' \triangleright N'$ *and* $\text{FN}(N) \subseteq L$ *then* $\text{FN}(N') \subseteq L'$.

Proof: We firstly prove a technical lemma.

> **Lemma A.2.** *If* $L \triangleright N \rightarrowtail L' \triangleright N'$ *and* $\text{FN}(N) \subseteq L$, *then* $L \subseteq L'$ *and* $\text{FN}(N') - \text{FN}(N) = L' - L$.
>
> **Proof:** That $L \subseteq L'$ immediately follows from the definition of the reduction rules because the set of localities in a configuration never decreases along reductions. To show that $\text{FN}(N') - \text{FN}(N) = L' - L$, we reason by induction on the length of the proof of $L \triangleright N \rightarrowtail L' \triangleright N'$. The only significant base case is when rule (NEW) is used: in such case, L' is obtained by adding to L the newly created locality l', which is the only locality in $\text{FN}(N') - \text{FN}(N)$. The inductive step is straightforward; when considering rule (STRUCT), notice that, if $N \equiv N'$, then $\text{FN}(N) = \text{FN}(N')$.

Now, $\text{FN}(N') \cap \text{FN}(N) \subseteq \text{FN}(N)$. Moreover, notice that $\text{FN}(N) - \text{FN}(N')$ might be not empty because some localities occurring in N but not as addresses of network nodes may disappear in N' due to inter-process communication. Hence, from Lemma A.2 we get $\text{FN}(N') = (\text{FN}(N') - \text{FN}(N)) \cup (\text{FN}(N') \cap \text{FN}(N)) \subseteq (L' - L) \cup \text{FN}(N) \subseteq (L' - L) \cup L' = L'$. □

Proposition 4.3. *If* N *is well-formed and* $L \triangleright N \rightarrowtail^* L' \triangleright N'$ *for* $\text{FN}(N) \subseteq L$, *then* N' *is well-formed.*

Proof: It is easy to prove, by induction on the rules, that the structural congruence $\equiv$ preserves well-formedness of nets. Thus, we are only left to prove that the

reduction relation does never transform a well-formed net into a net where two distinct nodes have the same address (indeed, the reduction rules could also be applied to nets that do not satisfy this property). To this aim, we first prove a Lemma stating that a single reduction step from a net N preserves the number of nodes having the same address. This property is expressed by using $clone(N)$ to denote the least number of nodes that should be removed from N to yield a well-formed net. To formally define function $clone(\cdot)$, we exploit the auxiliary function $mnl(\cdot)$ (mnl stands for 'multiset of node localities'), that, when applied to a net, returns the multiset of localities naming the nodes of the net. It is inductively defined over the syntax of nets as follows:

$$mnl(l ::^{\delta} C) \triangleq \{\!|l|\!\} \qquad\qquad mnl(N_1 \parallel N_2) \triangleq mnl(N_1) \uplus mnl(N_2)$$

where $\{\!|l_0, \ldots, l_n|\!\}$ denotes the multiset with elements $l_0, \ldots, l_n$ and $\uplus$ denotes multiset union. Now, for any μKLAIM net N, we can define $clone(N)$ as the cardinality of the multiset obtained by removing from $mnl(N)$ one occurrence of each different locality occurring in it.

Lemma A.3. *If $L \triangleright N \rightarrowtail L' \triangleright N'$ then $clone(N) = clone(N')$.*

Proof: We reason by induction on the length of the proof of the reduction $L \triangleright N \rightarrowtail L' \triangleright N'$. The base case is with axioms (OUT), (EVAL), (IN), (READ), (NEW) or (REPL), and it is trivial. In the inductive case, we reason by case analysis on the last rule applied. The cases of rules (MARK), (PAR) and (STRUCT) easily follow by induction: it can be easily seen that $\equiv$ preserves $clone(\cdot)$. Suppose now that the last applied rule is (SPLIT) and let $L \triangleright N_1 \rightarrowtail L' \triangleright N_2$ be its premise. Then, due to the form of the nets involved in the rule, we have $clone(N_1) = clone(N) + 1$ and $clone(N_2) = clone(N') + 1$. Since the proof of $L \triangleright N_1 \rightarrowtail L' \triangleright N_2$ is shorter than that of $L \triangleright N \rightarrowtail L' \triangleright N'$, we can apply induction and deduce that $clone(N_1) = clone(N_2)$, from which it follows that $clone(N) = clone(N')$ that proves the thesis.

To conclude, note that a net N is well-formed if and only if $clone(N) = 0$. Hence, by using Lemma A.3 and by a straightforward induction on the length of reduction sequences, the thesis easily follows. □

B. Formal Definitions for the Variations of Section 7

Definitions for Section 7.1.1. The structural congruence is modified by replacing rules (Alpha) and (Abs) in Table 2 with rules

$$\frac{P \ =_\alpha \ P'}{l ::^\delta \{\!\{ P \}\!\}_{\delta'} \equiv l ::^\delta \{\!\{ P' \}\!\}_{\delta'}} \ (\text{Alpha}') \qquad \frac{}{l ::^\delta \mathcal{AC} \equiv l ::^\delta \mathcal{AC} \mid \{\!\{ \mathbf{nil} \}\!\}_{\delta'}} \ (\text{Abs}')$$

The rules for the reduction relation are in Table 8.

Definitions for Section 7.2.1. The checking of grantings now deletes the capabilities passed in the tuple and returns the capabilities left. Its formal definition updates Table 4 as follows:

$$\frac{\begin{array}{l}\mu = [l_i \mapsto \pi_i]_{i=1,\ldots,k} \\ \delta_1 = \delta_1', [l \mapsto \bigcup_{i=1}^k (\pi_i - \delta(l))] \\ \forall\, i = 1, \ldots, k \,.\, \pi_i \sqsubseteq_\Pi (\delta(l) \cup \delta_1(l))\end{array}}{[\![\, l : \mu \,]\!]_\delta^{\delta_1} = \delta_1'} \qquad \frac{\begin{array}{l}[\![\, t_1 \,]\!]_\delta^{\delta_1} = \delta_2 \\ [\![\, t_2 \,]\!]_\delta^{\delta_2} = \delta_3\end{array}}{[\![\, t_1, t_2 \,]\!]_\delta^{\delta_1} = \delta_3}$$

Also the definition of function *match* must be updated; its new formulation relies on the following modification of rule (M_2):

$$\frac{\pi \sqsubseteq_\Pi \delta(l') \cup \mu(l) \qquad \mu = \mu', [l \mapsto (\pi - \delta(l'))]}{match_l^\delta(!\, x : \pi, l' : \mu) = \langle [l' \mapsto (\pi - \delta(l'))], [l'/x], l' : \mu' \rangle}$$

where only the capabilities delivered by the tuple that are not already owned by the executing node are used to enrich the policy of the executing process (this is needed to avoid delivering the process capabilities already in δ). As concerns (M_1), it is modified to additionally return the tuple passed as second argument to function $match_l^\delta$, while (M_3) is modified to additionally return the tuple resulting from the concatenation of the two tuples returned by its premises.

The rules for the reduction relation are in Table 9.

Definitions for Section 7.2.3. The rules for the reduction relation are in Table 10, where, for any $u \in dom(\delta)$, we let $\delta^{\{l''\}}(u) = \{(c, \{l''\}) \ : \ c \in \delta(u)\}$. Function $pol(\gamma)$ yields a simple capability list δ by deleting from γ all capability annotations; moreover, $static(\gamma)$ denotes the annotated capability list obtained from γ by removing all the capabilities that have not been statically assigned.

$$\text{(Out}')\quad \frac{[\![\, t \,]\!]_{\delta[\delta_1]}}{l ::^{\delta} \{\!\{ \mathbf{out}(t)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; l ::^{\delta} \{\!\{ P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}'|\langle t \rangle}$$

$$\text{(Eval}')\quad \frac{\delta'[\delta_1] \vdash_{l'} Q \triangleright \underline{Q}}{l ::^{\delta} \{\!\{ \mathbf{eval}(Q)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; l ::^{\delta} \{\!\{ P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}'|\{\!\{ \underline{Q} \}\!\}_{\delta_1}}$$

$$\text{(In}')\quad \frac{match_l^{\delta[\delta_1]}(T,t) = \langle \delta'', \sigma \rangle}{l ::^{\delta} \{\!\{ \mathbf{in}(T)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \;\rightarrowtail\; l ::^{\delta} \{\!\{ P\sigma \}\!\}_{\delta_1[\delta'']} \parallel l' ::^{\delta'} \mathbf{nil}}$$

$$\text{(Read}')\quad \frac{match_l^{\delta[\delta_1]}(T,t) = \langle \delta'', \sigma \rangle}{l ::^{\delta} \{\!\{ \mathbf{read}(T)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \;\rightarrowtail\; l ::^{\delta} \{\!\{ P\sigma \}\!\}_{\delta_1[\delta'']} \parallel l' ::^{\delta'} \langle t \rangle}$$

$$\text{(New}')\quad \frac{l' \notin L}{L \triangleright l ::^{\delta} \{\!\{ \mathbf{newloc}(l' : \delta').P \}\!\}_{\delta_1} \;\rightarrowtail\; L \cup \{l'\} \triangleright l ::^{\delta} \{\!\{ P \}\!\}_{\delta_1[l' \mapsto \delta_1(l)]} \parallel l' ::^{\delta'} \mathbf{nil}}$$

$$\text{(Repl}')\quad l ::^{\delta} \{\!\{ *P \}\!\}_{\delta_1} \;\rightarrowtail\; l ::^{\delta} \{\!\{ P \mid *P \}\!\}_{\delta_1}$$

$$\text{(Mark}')\quad \frac{l' = tgt(a) \qquad \{ar(a)\} \sqsubseteq_{\Pi} \delta_1(l') \qquad l ::^{\delta} \{\!\{ a.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; N}{l ::^{\delta} \{\!\{ \underline{a}.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; N}$$

$$\text{(Split}'_1)\quad \frac{L \triangleright l ::^{\delta} \mathcal{AC}_1 \parallel l ::^{\delta} \mathcal{AC}_2 \parallel N \;\rightarrowtail\; L' \triangleright l ::^{\delta} \mathcal{AC}'_1 \parallel l ::^{\delta} \mathcal{AC}'_2 \parallel N'}{L \triangleright l ::^{\delta} \mathcal{AC}_1|\mathcal{AC}_2 \parallel N \;\rightarrowtail\; L' \triangleright l ::^{\delta} \mathcal{AC}'_1|\mathcal{AC}'_2 \parallel N'}$$

$$\text{(Split}'_2)\quad \frac{L \triangleright l ::^{\delta} \{\!\{ P \}\!\}_{\delta_1} \parallel l ::^{\delta} \{\!\{ Q \}\!\}_{\delta_1} \parallel N \;\rightarrowtail\; L' \triangleright l ::^{\delta} \{\!\{ P' \}\!\}_{\delta_2} \parallel l ::^{\delta} \{\!\{ Q \}\!\}_{\delta_1} \parallel N'}{L \triangleright l ::^{\delta} \{\!\{ P|Q \}\!\}_{\delta_1} \parallel N \;\rightarrowtail\; L' \triangleright l ::^{\delta} \{\!\{ P' \}\!\}_{\delta_2} \mid \{\!\{ Q \}\!\}_{\delta_1} \parallel N}$$

plus rules (Par) and (Struct) from Table 6.
$[\![\cdot]\!]_{\_}$ is defined in Table 4 and $match_l(\cdot,\cdot)$ is defined in Table 5

Table 8: Acquisition by processes: operational semantics

Acknowledgements. We thank the anonymous reviewers for fruitful comments that helped in improving the paper.

$$(\textsc{Out}'') \quad \frac{[\![\, t \,]\!]_{\delta}^{\delta_1} = \delta_1'}{l ::^{\delta} \{\!\{ \mathbf{out}(t)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; l ::^{\delta} \{\!\{ P \}\!\}_{\delta_1'} \parallel l' ::^{\delta'} \mathcal{AC}'|\langle t \rangle}$$

$$(\textsc{Eval}'') \quad \frac{\delta_1 = \delta_1', \delta_1'' \qquad \delta' \vdash_{l'} Q \triangleright \underline{Q}}{l ::^{\delta} \{\!\{ \mathbf{eval}(Q)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; l ::^{\delta} \{\!\{ P \}\!\}_{\delta_1'} \parallel l' ::^{\delta'} \mathcal{AC}'|\{\!\{ \underline{Q} \}\!\}_{\delta_1''}}$$

$$(\textsc{In}'') \quad \frac{match_l^{\delta[\delta_1]}(T, t) = \langle \delta'', \sigma, t' \rangle}{l ::^{\delta} \{\!\{ \mathbf{in}(T)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \;\rightarrowtail\; l ::^{\delta} \{\!\{ P\sigma \}\!\}_{\delta_1[\delta'']} \parallel l' ::^{\delta'} \mathbf{nil}}$$

$$(\textsc{Read}'') \quad \frac{match_l^{\delta[\delta_1]}(T, t) = \langle \delta'', \sigma, t' \rangle}{l ::^{\delta} \{\!\{ \mathbf{read}(T)@l'.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \;\rightarrowtail\; l ::^{\delta} \{\!\{ P\sigma \}\!\}_{\delta_1[\delta'']} \parallel l' ::^{\delta'} \langle t' \rangle}$$

$$(\textsc{Mark}'') \quad \frac{l' = tgt(a) \qquad \delta_1 = \delta_1', [l' \mapsto \{ar(a)\}] \qquad l ::^{\delta} \{\!\{ a.P \}\!\}_{\delta_1'} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; N}{l ::^{\delta} \{\!\{ \underline{a}.P \}\!\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{AC}' \;\rightarrowtail\; N}$$

plus rules (New′), (Repl′), ($\textsc{Split}'_1$) and ($\textsc{Split}'_2$) from Table 8
and rules (Par) and (Struct) from Table 6

Table 9: Consumption of capabilities: operational semantics

References

[1] M. Abadi. Logic in access control. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 228–233. IEEE Computer Society, 2003.

[2] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.

[3] M. Abadi and C. Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Security Symposium (NDSS'03)*. The Internet Society, 2003.

[4] P. Adão and C. Fournet. Cryptographically sound implementations for communicating processes. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2006.

$$(\textsc{Out}''') \quad \frac{[\![\, t \,]\!]_{pol(\gamma)}}{l ::^{\gamma} \mathbf{out}(t)@l'.P \parallel l' ::^{\gamma'} C' \;\rightarrowtail\; l ::^{\gamma} P \parallel l' ::^{\gamma'} C' | \langle t \rangle^{l}}$$

$$(\textsc{Eval}''') \quad \frac{pol(static(\gamma')) \vdash_{l'} Q \triangleright \underline{Q}}{l ::^{\gamma} \mathbf{eval}(Q)@l'.P \parallel l' ::^{\gamma'} C' \;\rightarrowtail\; l ::^{\gamma} P \parallel l' ::^{\gamma'} C' | \underline{Q}}$$

$$(\textsc{In}''') \quad \frac{match_{l}^{pol(\gamma)}(T, t) = \langle \delta, \sigma \rangle}{l ::^{\delta} \mathbf{in}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle^{l''} \;\rightarrowtail\; l ::^{\gamma[\delta^{\{l''\}}]} P\sigma \parallel l' ::^{\delta'} \mathbf{nil}}$$

$$(\textsc{Read}''') \quad \frac{match_{l}^{pol(\gamma)}(T, t) = \langle \delta, \sigma \rangle}{l ::^{\delta} \mathbf{read}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle^{l''} \;\rightarrowtail\; l ::^{\gamma[\delta^{\{l''\}}]} P\sigma \parallel l' ::^{\delta'} \langle t \rangle^{l''}}$$

$$(\textsc{Revoke}) \quad \frac{\gamma' = \gamma'', \delta^{\{l\}}}{l ::^{\gamma} \mathbf{revoke}(\delta)@l'.P \parallel l' ::^{\gamma'} C' \;\rightarrowtail\; l ::^{\gamma} P \parallel l' ::^{\gamma''} C'}$$

plus rules (New), (Repl), (Split), (Par) and (Struct) from Table 6, with γ in place of δ everywhere.
$[\![\cdot]\!]_{\_}$ is defined in Table 4 and $\widetilde{match}_{l}(\cdot, \cdot)$ is defined in Table 5

Table 10: Revocation of capabilities: operational semantics

[5] K. Arnold, E. Freeman, and S. Hupfer. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[7] S. Bandhakavi, W. Winsborough, and M. Winslett. A trust management approach for flexible policy management in security-typed languages. In *To appear in the Proc. of CSF*. IEEE Computer Society, 2008.

[8] 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.

[9] M. Blaze, J. Feigenbaum, and A. D. Keromytis. The role of trust man-

agement in distributed systems security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, number 1603 in LNCS, pages 185–210. Springer-Verlag, 1999.

[10] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[11] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Trans. Program. Lang. Syst.*, 26(1):57–124, 2004.

[12] M. Bugliesi and M. Giunti. Secure implementations of typed channel abstractions. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 251–262. ACM, 2007.

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

[14] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*, number 1378 of Lecture Notes in Computer Science, pages 140-155, Springer, 1998.

[15] G. Castagna, J. Vitek, and F. Z. Nardelli. The seal calculus. *Inf. Comput.*, 201(1):1–54, 2005.

[16] A. Chaudhuri. Dynamic access control in a concurrent object calculus. In *Proc. of CONCUR*, volume 4137 of *LNCS*, pages 263–278. Springer, 2006.

[17] A. Chaudhuri and M. Abadi. Secrecy by typing and file-access control. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 112–123. IEEE Computer Society, 2006.

[18] H. Chen and S. Chong. Owned policies for information security. In *Proc. of CSFW*, pages 126–138. IEEE Computer Society, 2004.

[19] Y.-H. Chu, J. Feigenbaum, B. A. LaMacchia, P. Resnick, and M. Strauss. Referee: Trust management for web applications. *Computer Networks*, 29(8-13):953–964, 1997.

[20] V.-L. Chung and C. S. MacDonald. The development of a distributed capability system for VLOS. In F. Lai and J. Morris, editors, *Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002)*, Melbourne, Australia, 2002.

[21] 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.

[22] M. Coppo, M. Dezani, E. Giovannetti, and R. Pugliese. Dynamic and Local Typing for Mobile Ambients. In *Proc. of IFIP-TCS'04*, pages 577–590. Kluwer, 2004.

[23] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 24–35. ACM Press, 1999.

[24] 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.

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

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

[27] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. IETF RFC 2693, Sept. 1999.

[28] R. Focardi, R. Lucchi, and G. Zavattaro. Secure shared data-space coordination languages: a process algebraic surveys. *Sci. Comput. Program.*, 63(1):3–15, 2006.

[29] C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[30] C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In G. C. Necula and P. Wadler, editors, *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 323–335, 2008.

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

[32] D. Gelernter. Multiple Tuple Spaces in Linda. In J. G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *LNCS*, pages 20–27, 1989.

[33] L. Gong. A secure identity-based capability system. In *IEEE Symposium on Security and Privacy*, pages 56–65, 1989.

[34] D. Gorla and R. Pugliese. Enforcing Security Policies via Types. In *Proc. of Security in Pervasive Computing (SPC'03)*, volume 2802 of *LNCS*, pages 88–103. Springer-Verlag, 2003.

[35] D. Gorla and R. Pugliese. Resource Acces and Mobility Control with Dynamic Privileges Acquisition. In *Proc. of ICALP'03*, volume 2719 of *LNCS*, pages 119–132. Springer-Verlag, 2003.

[36] R. Gorrieri, R. Lucchi, and G. Zavattaro. Supporting secure coordination in secspaces. *Fundam. Inf.*, 73(4):479–506, 2006.

[37] D. Hagimont and N. D. Palma. Non-functional capability-based access control in the java environment. In *8th Int. Conf. on Object-Oriented Information Systems*, volume 2425 of *LNCS*, pages 323–335. Springer, 2002.

[38] R. Handorean and G.-C. Roman. Secure sharing of tuple spaces in ad hoc settings. *Electr. Notes Theor. Comput. Sci.*, 85(3), 2003.

[39] R. R. Hansen, C. W. Probst, and F. Nielson. Sandboxing in myklaim. In *First International Conference on Availability, Reliability and Security (ARES)*, pages 174–181. IEEE Computer Society, 2006.

[40] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Trans. Program. Lang. Syst.*, 24(5):566–591, 2002.

[41] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.

[42] C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. of FoSSaCS'05*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[43] N. Li, B. N. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.

[44] M. Merro and M. Hennessy. A bisimulation-based semantic theory of safe ambients. *ACM Trans. Program. Lang. Syst.*, 28(2):290–330, 2006.

[45] M. Miller, K. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Systems Research Laboratory, 2003.

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

[47] F. Nielson, H. R. Nielson, and R. R. Hansen. Validating firewalls using flow logics. *Theor. Comput. Sci.*, 283(2):381–418, 2002.

[48] H. R. Nielson and F. Nielson. Shape analysis for mobile ambients. *Nord. J. Comput.*, 8(2):233–275, 2001.

[49] 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.

[50] 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.

[51] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. *J. Autom. Reasoning*, 31(3-4):335–370, 2003.

[52] A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.

[53] 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*, number 2000 in LNCS, pages 86–101. Springer, 2000.

[54] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[55] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. of CSF*, pages 203–217. IEEE Computer Society, 2007.

[56] Sun Microsystems. Javaspace specification. `http://java.sun.com/`, 1999.

[57] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proc. of CSFW*, pages 202–216. IEEE Computer Society, 2006.

[58] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 558–563. IEEE Computer Society, 1986.

[59] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, pages 179–193, 2004.

[60] N. I. Udzir, A. M. Wood, and J. L. Jacob. Coordination with multicapabilities. *Sci. Comput. Program.*, 64(2):205–222, 2007.

[61] M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 291–302, 1999.

[62] A. Wood. Coordination with attributes. In *COORDINATION '99: Proceedings of the Third International Conference on Coordination Languages and Models*, pages 21–36. Springer-Verlag, 1999.

[63] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998.