From Flow Logic to Static Type Systems for Coordination Languages*

Rocco De Nicola[1], Daniele Gorla[2], René Rydhof Hansen[3], Flemming Nielson[4], Hanne Riis Nielson[4], Christian W. Probst[4], and Rosario Pugliese[1]

[1] Dip. Sistemi e Informatica, Univ. di Firenze
[2] Dip. Informatica, Univ. di Roma "La Sapienza"
[3] Department of Computer Science, Aalborg University
[4] Informatics and Mathematical Modelling, Technical University of Denmark

Abstract. Coordination languages are often used to describe open ended systems. This makes it challenging to develop tools for guaranteeing security of the coordinated systems and correctness of their interaction. Successful approaches to this problem have been based on type systems with dynamic checks; therefore, the correctness properties cannot be statically enforced. By contrast, static analysis approaches based on Flow Logic usually guarantee properties statically. In this paper we show how to combine these two approaches to obtain a static type system for describing secure access to tuple spaces and safe process migration for a dialect of the language KLAIM.

1 Introduction

Coordination languages allow two or more components of an application to communicate, by reading/removing/adding data to a shared communication medium, in order to accomplish shared goals. These languages are often being used to program applications in open ended systems, namely systems whose overall structure can change dynamically in unpredictable ways because the entities involved can join and leave at any time. This open nature exposes applications/systems to malicious accesses to their data/resources. Also, when process mobility is permitted, one can easily conceive trojan horses or viruses spawned at remote localities by malicious entities.

This scenario makes it challenging to develop tools for guaranteeing security of coordinated components and correctness of their interaction. Discretionary *access control* mechanisms have been then designed based either on specifying the permitted operations associated to the objects, or on specifying the *capabilities* that the different subjects have on the objects. The capability-based approach appears to be more appropriate than the access-control one for open distributed systems (see e.g. [12]), because capabilities can be distributed to the subjects, rather than being attached to the objects, and can be passed on. Moreover, their different categories need not to be statically fixed.

Different techniques have also been devised to enforce access control (see e.g. [11]). The most traditional one is based on a *reference monitor* that dynamically intercepts

* This work has been supported by the EU project SENSORIA, IST-2005-016004.

each attempted access to a (critical) resource and determines whether the intended operations should be allowed or denied. The main disadvantage of this approach is that security properties can only be checked dynamically, thus lowering the performance of systems. In order to limit these drawbacks, many *static analysis* techniques [8] have been devised. These techniques originate from the work on compilers [1] where it is imperative that all relevant behaviour of systems be determined statically. The result of analysing a program is an analysis estimate that gives a *global summary* of the properties of interest. However, these approaches require a knowledge of the full system and make the analysis more difficult.

To overcome all these limitations, hybrid approaches have been investigated that take advantage of both static and dynamic checks. This is, e.g., the case of the capability-based type systems for KLAIM (*Kernel Language for Agents Interaction and Mobility*, [2]), an experimental language specifically designed to program distributed systems made up of several mobile components. KLAIM has proved to be suitable for programming a wide range of distributed applications with agents and code mobility. Its primitives allow programmers to distribute/retrieve data and processes to/from the nodes of a net and extend Linda's notion of *generative communication* [4] through multiple shared tuple spaces.

In the capability-based type systems for KLAIM (see e.g. [3, 5]), capabilities are used to specify the access control policies stating which operations (**in**, **out**, **eval**, ...) 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. Access requests are mostly checked statically, but some dynamic type check is used to deal with data communication and process migration. In the former case, the dynamic checks are needed because no constraint is put on the kind of data inserted in tuple spaces; hence, withdrawal of data must be type controlled to establish matching with the input pattern. In the latter case, the type check has to be deferred to run-time because the target node of a process migration, and, hence, its policy, could be statically unknown.

In this paper we show how to use ideas from the Flow Logic approach [10] to static analysis to enhance KLAIM's type systems with means for giving a global account of the behaviour of the system. Indeed, this seems necessary in order to deal with the distributed nature of tuple spaces; furthermore, it allows us to develop a fully static type system. On the other hand, the Flow Logic approach borrows from the type-based approach in being compositional in axiomatising when analysis estimates are valid for a given system (although the actual computation of the best, i.e. least, analysis estimate requires global solution of a system of constraints [9]).

The rest of the paper is structured as follows. In Section 2 we introduce the syntax and semantics of the dialect of KLAIM considered; we dispense with an operation for creating new localities but instead use a primitive for accepting processes from the environment. A Flow Logic for the language is developed in Section 3 and used as inspiration to design the fully static type system presented in Section 4. Our major results, stated in Sections 3 and 4, prove that the two analysis techniques are in accordance. We conclude in Section 5.

2 A Dialect of KLAIM

Syntax. The process calculus used here, like other members of the KLAIM family, consists of three layers: nets, processes, and actions. Nets specify the overall structure of a system, including where processes and tuple spaces are located. Processes are the actors in this system and execute by performing actions. The syntax for all these components is presented in the upper part of Figure 1, whereas in the lower part it is reported the syntax of the capability-based types.

NETS		LOCALITIES	
$N ::= l ::^{e\delta} P$	process	$\ell ::= l$	locality constant
$\mid\ l :: \langle et \rangle$	located tuple	$\mid$ **self**	self
$\mid\ N_1 \parallel N_2$	net composition	$\mid\ u$	locality variable
PROCESSES		TEMPLATES	
$P ::=$ **nil**	empty process	$T ::= \ell$	locality
$\mid\ \alpha.P$	action prefixing	$\mid\ !u$	input variable
$\mid\ P_1 \mid P_2$	parallel composition	$\mid\ \ell, T$	multiple fields
$\mid\ *P$	replication	$\mid\ !u, T$	multiple fields
ACTIONS		TUPLES	
$\alpha ::=$ **out**$(t)@\ell$	output	$t ::= \ell$	element
$\mid$ **in**$(T)@\ell$	input	$\mid\ \ell, t$	multiple elements
$\mid$ **read**$(T)@\ell$	read	EVALUATED TUPLES	
$\mid$ **eval**$(P : \delta)@\ell$	migration	$et ::= l$	evaluated element
$\mid$ **accept**(δ)	admission	$\mid\ l, et$	multiple evaluated elements

Capabilities	Policies	EvaluatedPolicies
$\{o, i, r, e, a\}$	$\delta : \mathsf{Loc} \cup \{\mathbf{self}\} \to \mathcal{P}(\mathsf{Capabilities})$	$e\delta : \mathsf{Loc} \to \mathcal{P}(\mathsf{Capabilities})$

Fig. 1. Syntax of KLAIM

A net consists of processes or tuples located at a locality l, or a composition of two nets. Processes are built up from the special process **nil**, that does not perform any action (and is often omitted), and from the basic actions by means of prefixing, parallel composition and replication. hence, the actual building blocks of processes are actions: **out** and **in** actions permit to produce/withdraw tuples to/from a possibly remote tuple space; **read** is a non-destructive variant of **in**; **eval** models mobility by spawning processes from a locality to another one, where it will be evaluated; **accept** allows processes coming from the environment to get into the system. In fact, **accept**, first introduced in [6], makes the language more suitable to model *open* systems.

As regards the tuples used for communication, we distinguish between *tuples* and *evaluated tuples*. An evaluated tuple is a sequence of values, that in our case are element of the set Loc of localities, and can be stored in tuple spaces. In contrast, tuples are allowed to contain variables and self-references denoted by **self**. Tuples are used in pro-

$$match(l, l) = \epsilon \qquad match(!u, l) = [u \mapsto l] \qquad \frac{match(T_1, et_1) = \sigma_1 \quad match(T_2, et_2) = \sigma_2}{match((T_1, T_2), (et_1, et_2)) = \sigma_1 \circ \sigma_2}$$

Fig. 2. Matching function

cesses to compose data to be communicated. When inputting tuples from tuple spaces, processes need to be able to select which tuple should be read or input. This filtering is performed by means of *templates*, that are similar to tuples, but can also contain *input variables* denoted as $!u$. In the latter case, u is *bound* in the continuation process and will be used to retrieve information dynamically (u will be replaced with some locality in the continuation process upon successful matching of the template against a tuple – see function *match* in Figure 2). A variable that is not bound is called *free*.

Network nodes are equipped with a *policy* that expresses the discretionary access control policy that should be enforced upon the system. As usual, a discretionary access control policy states which *subjects* can access which *objects* using what *capabilities*. Here we take *subjects* to be the localities where the action is executed, *objects* to be the localities accessed (for example, placing a new evaluated tuple there, inputting or reading an evaluated tuple, or spawning a new process), and *capabilities* to be indicators of the access operation, i.e., elements of the set Capabilities representing the out-, in-, read-, eval-, and accept-capability respectively. Policies are represented as *capability lists*. Thus, a policy, placed at some locality l_s, maps an object locality l_o to the set of capabilities with which the subject l_s can access l_o. Formally, we distinguish between Policies and EvaluatedPolicies. They both are functions from localities to sets of capabilities and differ only in whether they allow **self** to be used as a locality. Policies δ embedded in the syntax can use **self**, whereas (evaluated) policies $e\delta$ placed at some locality, written $l ::^{e\delta} \ldots$, may not.

Semantics. The semantics is an operational semantics in the form of a reduction semantics. It makes use of the function *match*, defined in Figure 2, for performing the variable bindings when reading or inputting. The matching proceeds by comparing a template T componentwise with an evaluated tuple et. There are two possibilities for the match to succeed. Either both the template and the tuple begin with the same locality, or the template begins with an input variable. The result of a successful match is a substitution[1] that replaces the template's input variables with the values that occurred at corresponding positions in the evaluated tuple. In the sequel, we shall assume that templates T are well-formed in the sense that they do not contain both u and $!u$, and do not contain multiple occurrences of $!u$ for the same locality variable u.

The reduction semantics operates on closed processes, i.e. processes without free variables, but it still needs to take care of the occurrences of **self**. This is achieved by two auxiliary functions that map tuples (without free locality variables) to evaluated tuples, and policies to evaluated policies, respectively. They are both indexed with the

[1] As usual, 'ϵ' denotes the empty substitution and '$\circ$' denotes composition of substitutions with disjoint domains.

$$N_1 \,\|\, N_2 \;\equiv\; N_2 \,\|\, N_1 \qquad (N_1 \,\|\, N_2) \,\|\, N_3 \;\equiv\; N_1 \,\|\, (N_2 \,\|\, N_3)$$

$$l ::^{e\delta} P \;\equiv\; l ::^{e\delta} (P \,|\, \mathbf{nil}) \qquad l ::^{e\delta} (P_1 \,|\, P_2) \;\equiv\; l ::^{e\delta} P_1 \,\|\, l ::^{e\delta} P_2 \qquad l ::^{e\delta} *P \;\equiv\; l ::^{e\delta} P \,|\, *P$$

Fig. 3. Structural Congruence

locality to be used instead of **self** and we shall allow to use the same syntax for both.

$$(\!|\cdot|\!)_l : (\mathsf{Loc} \cup \{\mathbf{self}\}) \to \mathsf{Loc} \text{ given by } (\!|\ell|\!)_l = \begin{cases} l & \text{if } \ell = \mathbf{self} \\ l' & \text{if } \ell = l' \in \mathsf{Loc} \end{cases}$$

$$(\!|\cdot|\!)_l : \mathsf{Policy} \to \mathsf{EvaluatedPolicy} \text{ given by } (\!|\delta|\!)_l(l') = \bigcap \{\delta(\ell) |\, (\!|\ell|\!)_l = l'\}$$

The first function simply replaces any occurrence of **self** with the subscript, which is supposed to denote the intended meaning of **self**. We trivially extend it from working on single localities to working on sequences in a componentwise manner. We also trivially extend it to work on templates (without free locality variables) by defining it to act as the identity on input variables. The second function gives $(\!|\delta|\!)_l(l') = \delta(l')$ except when $l' = l$ in which case it gives $(\!|\delta|\!)_l(l) = \delta(l) \cap \delta(\mathbf{self})$ meaning that both the policies of l and **self** are imposed.

Figure 4 shows the semantics for our calculus. In the reduction rules, we use L to keep track of used localities and test if a given locality exists. The formulae of the form $\mathsf{RM}[\cdots]$ correspond to the checks that the (eventually superfluous) reference monitor must perform and make the intentions of the security policy clear. As an example, for the output action the formula $\mathsf{RM}[e\delta(l') \ni o]$ is intended to ensure that the local policy, $e\delta$, does indeed allow output to the locality l'. As usual, reductions are given compositionally and up-to a (quite standard) structural congruence, defined in Figure 3.

The **out** action takes an evaluated tuple and outputs it at the tuple space identified by ℓ. Note that, as for all other actions, execution of the current subprocess is stuck if the tuple is not fully evaluated, that is if it still contains variables. The **in** action takes a template T and a locality ℓ, and uses the judgement for *match* previously defined to select a tuple from the tuple space at ℓ by matching all tuples against T. As an effect of the **in** action, the matched tuple is removed from the tuple space and the substitution σ computed by *match* is applied to the rest of the process, thereby substituting input variables in T by the values bound to them. The **eval** action sends its argument Q for evaluation to the locality identified by ℓ; the policy used is the evaluated version of the one specified and our static analysis techniques will ensure that such a policy conforms to the policy specified for the target node. The **accept** action admits into a system new processes coming from the environment. In case of dynamic enforcement, i.e. using reference monitors, this rule is as straightforward as the rest, since the behaviour of incoming processes is checked dynamically during their execution. In case of static enforcement, as is the focus of the present paper, we need to ensure that a new process, Q, is only admitted if it satisfies sufficiently strong guarantees that have been used in validating the known part of the system. We use the formula $\overline{\mathsf{RM}}[\phi_{acc}]$ to express this. We will need to postpone the explanation of the formula ϕ_{acc} used until after the two static analysis techniques have been developed. Intuitively, it will ensure that the

$$\frac{(\!|\ell|\!)_l = l' \in L \qquad (\!|t|\!)_l = et \qquad \mathsf{RM}[e\delta(l') \ni o]}{L \triangleright l ::^{e\delta} \mathbf{out}(t)@\ell.P \longrightarrow l ::^{e\delta} P \,\|\, l' :: \langle et \rangle}$$

$$\frac{(\!|\ell|\!)_l = l' \qquad match((\!|T|\!)_l, et) = \sigma \qquad \mathsf{RM}[e\delta(l') \ni i]}{L \triangleright l ::^{e\delta} \mathbf{in}(T)@\ell.P \,\|\, l' :: \langle et \rangle \longrightarrow l ::^{e\delta} P\sigma}$$

$$\frac{(\!|\ell|\!)_l = l' \qquad match((\!|T|\!)_l, et) = \sigma \qquad \mathsf{RM}[e\delta(l') \ni r]}{L \triangleright l ::^{e\delta} \mathbf{read}(T)@\ell.P \,\|\, l' :: \langle et \rangle \longrightarrow l ::^{e\delta} P\sigma \,\|\, l' :: \langle et \rangle}$$

$$\frac{(\!|\ell|\!)_l = l' \in L \qquad (\!|\delta'|\!)_l = e\delta' \qquad \mathsf{RM}[e\delta(l') \ni e]}{L \triangleright l ::^{e\delta} \mathbf{eval}(Q : \delta')@\ell.P \longrightarrow l ::^{e\delta} P \,\|\, l' ::^{e\delta'} Q}$$

$$\frac{(\!|\delta'|\!)_l = e\delta' \qquad \mathsf{RM}[e\delta(l) \ni a] \qquad \overline{\mathsf{RM}}[\phi_{acc}]}{L \triangleright l ::^{e\delta} \mathbf{accept}(\delta').P \longrightarrow l ::^{e\delta} P \,\|\, l ::^{e\delta'} Q}$$

$$\frac{L \triangleright N_1 \longrightarrow N_1'}{L \triangleright N_1 \,\|\, N_2 \longrightarrow N_1' \,\|\, N_2} \qquad \frac{N \equiv N_1 \qquad L \triangleright N_1 \longrightarrow N_2 \qquad N_2 \equiv N'}{L \triangleright N \longrightarrow N'}$$

Fig. 4. Operational Semantics of KLAIM

incoming process (viz., Q) respects the specified policy δ' that, in turn, respects the policy $e\delta$ of the node where the action is performed (viz., l).

As we stated in the Introduction, there are two main approaches to enforce a given access control policy on a system: one is to check it dynamically by means of a reference monitor; the other is to develop a static analysis technique. Subsequently, we shall refer to the reference monitor semantics by writing $L \triangleright N \longrightarrow_{\mathsf{on}} N'$. It is specified as in Figure 4 by letting $\mathsf{RM}[\phi]$ mean ϕ and $\overline{\mathsf{RM}}[\phi]$ mean *true* (and so can be removed). Similarly, we shall refer to the semantics without reference monitors by writing $L \triangleright N \longrightarrow_{\mathsf{off}} N'$. It is specified as in Figure 4 by letting $\mathsf{RM}[\phi]$ mean *true* (and so can be removed) and $\overline{\mathsf{RM}}[\phi]$ mean ϕ.

Running Example. As a running example, throughout the paper we will consider a scenario where a user wants to collect and elaborate some pieces of information, e.g., electronic books, scattered on network nodes. The user can exploit both remote operations and process migration, e.g., to deal with possible network failures. The KLAIM net modelling the scenario includes the user process (located at l_U), a directory service process (located at l_D), and some data containers; for simplicity sake, we consider only two data containers, located at l_{C1} and l_{C2}. Moreover, in the example we assume that some sort of primitive data, like e.g. strings, are available and can be used as fields of data tuples. This is only for convenience, since all of the primitive data can be encoded in terms of localities.

$$l_U ::^{e\delta_U} P_U \,\|\, l_D ::^{e\delta_D} P_D \,\|\, l_{C1} ::^{e\delta_{C1}} \mathbf{nil} \,\|\, l_{C2} ::^{e\delta_{C2}} \mathbf{nil}$$
$$\|\, l_D :: \langle \text{library}, l_{C1} \rangle | \langle \text{library}, l_{C2} \rangle \,\|\, l_{C1} :: \langle \text{J.R.R. Tolkien, The Hobbit} \rangle$$

Each node hosts running processes that must obey a given access policy and/or contains data tuples. The processes are defined as

$$
\begin{aligned}
P_U &= \mathbf{eval}(P_1 : \delta)@l_D. * \mathbf{in}(!source, !data)@l_U. < elaborate\ data > \\
P_1 &= *\mathbf{read}(\text{library}, !u)@l_D.\mathbf{eval}(P_2 : \delta)@u \\
P_2 &= \mathbf{read}(\text{J.R.R. Tolkien}, !title)@\mathbf{self}.\mathbf{out}(\mathbf{self}, title)@l_U \\
P_D &= \mathbf{accept}(\delta_a)
\end{aligned}
$$

and use the policies

$$
\begin{aligned}
\delta &= [l_U \mapsto \{o\}, l_D \mapsto \{r\},\ l_{C1} \mapsto \{e, r\},\ l_{C2} \mapsto \{e, r\}] \\
\delta_a &= [l_D \mapsto \{r\},\ l_{C1} \mapsto \{e, r\},\ l_{C2} \mapsto \{e, r\}]
\end{aligned}
$$

while the access policies of nodes are the following ones:

$$
\begin{aligned}
e\delta_U &= [l_U \mapsto \{i\},\ l_D \mapsto \{e\}] \\
e\delta_D &= [l_D \mapsto \{r, a\},\ l_{C1} \mapsto \{e\},\ l_{C2} \mapsto \{e\}] \\
e\delta_{C1} &= [l_{C1} \mapsto \{r\},\ l_U \mapsto \{o\}] \\
e\delta_{C2} &= [l_{C2} \mapsto \{r\},\ l_U \mapsto \{o\}]
\end{aligned}
$$

For completeness sake, we note that the example is intended to illustrate how the calculus and, in later sections, the analysis and the type system work. It is not meant to be a complete specification of a distributed system and therefore it does not include modelling of, e.g., scheduling and similar concepts.

3 Flow Logic

We shall now develop an analysis that captures the behaviour of nets. The analysis computes an *over-approximation* of the actual behaviour of a KLAIM net. We first present the abstract domains underlying the analysis and next define the judgements for nets, processes, actions and matchings. We conclude this section by analysing our running example and presenting the theoretical properties of our approach.

Analysis Domains. We shall use the following analysis domains:

- $\hat{T} \in \mathsf{Loc} \to \mathcal{P}(\mathsf{Loc}^*)$ is an *abstract tuple space*; it is an over-approximation of the set of all tuples (of locality constants) that may at some point reside in the tuple space of a given locality constant.
- $\hat{\sigma} \in \mathsf{LocVar} \to \mathcal{P}(\mathsf{Loc})$ is an *abstract environment*; it keeps a record of all locality constants that a given locality variable might at some point be bound to. (This functionality suffices because the structural congruence does not contain α-renaming of bound variables.)
- $\partial \in \mathsf{AbstractPolicy} = \mathsf{Loc} \to \mathcal{P}(\mathsf{Capabilities})$ is an *abstract policy* somewhat like the concrete policy $\delta \in \mathsf{Policy}$; however, it takes the *union* of possibilities rather than the intersection because it is descriptive rather than prescriptive. Abstract policies form a lattice based on the natural ordering on partial functions, written $\sqsubseteq$, i.e. $\partial \sqsubseteq \partial'$ if and only if $\mathrm{dom}(\partial) \subseteq \mathrm{dom}(\partial')$ and $\partial(l) \subseteq \partial'(l)$, for every $l \in \mathrm{dom}(\partial)$.

$$\frac{(\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} P : \partial', \varrho' \qquad \partial' \setminus_{\{l\}} e\delta \sqsubseteq \varrho \qquad \Delta(l) \setminus_{\{l\}} e\delta \sqsubseteq \varrho \qquad \varrho' \sqsubseteq \varrho}{(\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{N}} l ::^{e\delta} P : \varrho}$$

$$\frac{\{et\} \subseteq \hat{T}(l)}{(\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{N}} l :: \langle et \rangle : \varrho} \qquad \frac{(\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{N}} N_1 : \varrho \qquad (\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{N}} N_2 : \varrho}{(\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{N}} N_1 \,\|\, N_2 : \varrho}$$

Fig. 5. Static Analysis of Nets

- $\Delta \in \mathsf{Loc} \to \mathsf{AbstractPolicy}$ is a record of policies for remotely evaluated processes.
- $\varrho \in \mathsf{Loc} \to \mathsf{AbstractPolicy}$ is a *record of violations of policies*. It records all the actions that may have been performed during the evolution of the net and that were *not* permitted by the local policy; the first argument is the subject locality where the action was initiated, and the second argument is the object locality where the action had effect, and the resulting set of capabilities are the offending ones. Hence a program will only be acceptable if it can be analysed with $\varrho = \bot$.
- $\Lambda \in \mathcal{P}(\mathsf{Loc})$ is a set of localities of interest at a given point. In general, we shall analyse processes at sets of localities (rather than a single locality) in order to obtain a context insensitive analysis. A context sensitive analysis, i.e., a more precise analysis, can be obtained by analysing processes at single localities.

Analysis of Nets. The judgement for the analysis of a net N has the form

$$(\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{N}} N : \varrho$$

and is defined by the inference system of Figure 5. As is usual in Flow Logic, we provide a componentwise definition.

To determine the potential violations of the policy for a located process, we use the following auxiliary notation for "subtracting" two policies:

$$\begin{aligned} \partial_1 \setminus_\Lambda \partial_2 \quad &: \mathsf{Loc} \to \mathsf{AbstractPolicy} \\ (\partial_1 \setminus_\Lambda \partial_2)(\lambda_s)(\lambda_o) &= \begin{cases} \partial_1(\lambda_o) \setminus \partial_2(\lambda_o) & \text{if } \lambda_s \in \Lambda \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

Analysis of Processes. The judgement for the analysis of a process P has the form

$$(\hat{T},\Delta,\hat{\sigma}) \models_{\mathrm{P}}^{\Lambda} P : \partial, \varrho$$

and is defined by the inference system of Figure 6. The intention is that when true, the components $\hat{T}$, Δ, $\hat{\sigma}$, ∂ and ϱ correctly capture not only the behaviour of the process P (when located at one of the localities $\lambda \in \Lambda$) but also the behaviour of all the processes it may evolve into. Any violation encountered during analysis of the process is recorded in ϱ, whereas ∂ approximates the actual policy employed by the process. The definition is fairly straightforward in that it inspects the components of a process in a structural way making use of the judgement for actions to be introduced next.

$$(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} \mathbf{nil} : \partial,\varrho \qquad \frac{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P_1 : \partial,\varrho \qquad (\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P_2 : \partial,\varrho}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P_1 \,|\, P_2 : \partial,\varrho}$$

$$\frac{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P : \partial,\varrho}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} *P : \partial,\varrho} \qquad \frac{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P : \partial,\varrho \qquad (\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \alpha : \partial,\varrho}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} \alpha.P : \partial,\varrho}$$

Fig. 6. Static Analysis of Processes

$$\frac{(\!|t|\!)^{\Lambda}_{\hat{\sigma}} \subseteq \hat{T}\langle (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \rangle \qquad [\,(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \to \{o\}\,] \sqsubseteq \partial}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \mathbf{out}(t)@\ell : \partial,\varrho} \qquad \frac{\hat{\sigma} \models^{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}}_{1} T : \hat{T}[(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}] \triangleright \hat{W} \qquad [\,(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \to \{i\}\,] \sqsubseteq \partial}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \mathbf{in}(T)@\ell : \partial,\varrho}$$

$$\frac{(\!|\delta|\!)^{\Lambda} \sqsubseteq \partial \qquad [\Lambda \to \{a\}\,] \sqsubseteq \partial}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \mathbf{accept}(\delta) : \partial,\varrho} \qquad \frac{\hat{\sigma} \models^{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}}_{1} T : \hat{T}[(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}] \triangleright \hat{W} \qquad [\,(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \to \{i\}\,] \sqsubseteq \partial}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \mathbf{read}(T)@\ell : \partial,\varrho}$$

$$\frac{(\hat{T},\Delta,\hat{\sigma}) \models^{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}}_{\mathrm{P}} P : \partial',\varrho \quad \forall \lambda \in (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} : (\!|\delta|\!)^{\Lambda} \sqsubseteq \Delta(\lambda) \quad \partial' \setminus_{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}} (\!|\delta|\!)^{\Lambda} \sqsubseteq \varrho \quad [\,(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \to \{e\}\,] \sqsubseteq \partial}{(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \mathbf{eval}(P : \delta)@\ell : \partial,\varrho}$$

Fig. 7. Static Analysis of Actions

Analysis of Actions. The judgement for the analysis of an action α has the form

$$(\hat{T},\Delta,\hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \alpha : \partial,\varrho$$

and is defined by the inference system of Figure 7.

To transform localities $\ell \in \mathsf{Loc} \cup \{\mathbf{self}\} \cup \mathsf{LocVar}$ into the set of localities that they denote, we make use of the auxiliary function

$$(\!|\cdot|\!)^{\Lambda}_{\hat{\sigma}} \;:\; \mathsf{Loc} \cup \{\mathbf{self}\} \cup \mathsf{LocVar} \to \mathcal{P}(\mathsf{Loc})$$
$$(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} = \begin{cases} \{\ell\} & \text{if } \ell \in \mathsf{Loc} \\ \Lambda & \text{if } \ell = \mathbf{self} \\ \hat{\sigma}(\ell) & \text{if } \ell \in \mathsf{LocVar} \end{cases}$$

This transformation is straightforward for locality constants, while it exploits the set Λ of locality constants that **self** might stand for, in the case of **self**, and the abstract environment $\hat{\sigma}$, in the case of locality variables. This operation is extended to tuples t by taking the cartesian product of all components. For evaluated tuples et, we have $(\!|et|\!)^{\Lambda}_{\hat{\sigma}} = \{et\}$.

To transform concrete policies into abstract policies, we make use of the auxiliary function

$$(\!|\delta|\!)^{\Lambda} \;:\; \mathsf{AbstractPolicy}$$
$$(\!|\delta|\!)^{\Lambda}(\lambda) = \bigcap \{\delta(\ell) |\, \lambda \in (\!|\ell|\!)^{\Lambda}_{\perp}, \ell \in \mathsf{Loc} \cup \{\mathbf{self}\}\}$$

This operation is somewhat reminiscent of the way tuples were transformed into evaluated tuples. Since (concrete) policies are not defined on locality variables, it suffices using the empty abstract environment $\bot$ in the conversion of localities. For evaluated policies, we have $(\!|e\delta|\!)^{\Lambda}(l) = e\delta(l)$.

To more easily express that the appropriate record of actions is captured by the policy component ∂, we use the notation

$$\begin{aligned}[X \to Y] \quad &: \mathsf{Loc} \to \mathcal{P}(\mathsf{Capability}) \\ [X \to Y](\lambda) &= \begin{cases} Y \text{ if } \lambda \in X \\ \emptyset \text{ otherwise} \end{cases}\end{aligned}$$

where λ denotes the locality constant where the action might have effect and Y usually is a singleton set. In the case of **out**, **in**, **read** and **eval**, we take X to be the set $(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}$; in the case of **accept**, we take X to be the set Λ of current localities.

Since most of the rules need to take effect for any element in some set X of locality constants, it is frequently necessary to write logical formulae using universal and existential quantifiers. The resulting formulae tend to clutter the understanding of the more subtle features of the Flow Logic specification and we have therefore decided to introduce two notational shorthands so as to reduce the explicit use of quantifiers. The notations are formally defined by:

$$\begin{aligned}\Psi[X] &= \textstyle\bigcup_{x \in X} \Psi(x) = \{z \mid \exists x \in X : z \in \Psi(x)\} \\ \Psi\langle X \rangle &= \textstyle\bigcap_{x \in X} \Psi(x) = \{z \mid \forall x \in X : z \in \Psi(x)\}\end{aligned}$$

It is worth pointing out that this permits to use them in inclusions and that they can be expanded away using the following tautologies:

$$\begin{aligned}\Psi[X] \subseteq Z &\iff \forall x \in X\colon \Psi(x) \subseteq Z \\ Z \subseteq \Psi\langle X \rangle &\iff \forall x \in X\colon Z \subseteq \Psi(x)\end{aligned}$$

As an example, in the rule for $\mathbf{out}(t)@\ell$ the premise $(\!|t|\!)^{\Lambda}_{\hat{\sigma}} \subseteq \hat{T}\langle(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}\rangle$ expresses that *all* the values that t may evaluate to are included in *all* the tuple spaces that could be associated with the locality ℓ.

Analysis of Matching. The auxiliary judgement

$$\hat{\sigma} \models^{\Lambda}_{i} T : \hat{U} \triangleright \hat{W}$$

defined by the inference system of Figure 8 is used in the rules for $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$ in Figure 7 to ensure that the matching may succeed. The set of tuples of interest are those of the tuple space of ℓ, that is, $\hat{T}[(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}]$. The judgement expresses that matching should start at position i in the template T, $\hat{U}$ contains the set of tuples that we are matching against, $\hat{W}$ contains the tuples from $\hat{U}$ that successfully match T from position i and onwards, and $\hat{\sigma}$ records the appropriate bindings that need to be performed. In the rules of Figure 7, $\pi_i(et)$ denotes the i'th component of the tuple et and $\pi_i(\hat{V})$ is the componentwise extension of the operation to sets of tuples.

$$\frac{\{et \in \hat{U} \mid \pi_i(et) \in (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \ \wedge\ |et| = i\} \subseteq \hat{W}}{\hat{\sigma} \models^{\Lambda}_i \ell : \hat{U} \triangleright \hat{W}} \qquad \frac{\{et \in \hat{U} \mid |et| = i\} \subseteq \hat{W} \qquad \pi_i(\hat{W}) \subseteq \hat{\sigma}(u)}{\hat{\sigma} \models^{\Lambda}_i !u : \hat{U} \triangleright \hat{W}}$$

$$\frac{\{et \in \hat{U} \mid \pi_i(et) \in (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \ \wedge\ |et| \geq i\} \subseteq \hat{V} \qquad \hat{\sigma} \models^{\Lambda}_{i+1} T : \hat{V} \triangleright \hat{W}}{\hat{\sigma} \models^{\Lambda}_i \ell, T : \hat{U} \triangleright \hat{W}}$$

$$\frac{\{et \in \hat{U} \mid |et| \geq i\} \subseteq \hat{V} \qquad \hat{\sigma} \models^{\Lambda}_{i+1} T : \hat{V} \triangleright \hat{W} \qquad \pi_i(\hat{W}) \subseteq \hat{\sigma}(u)}{\hat{\sigma} \models^{\Lambda}_i !u, T : \hat{U} \triangleright \hat{W}}$$

Fig. 8. Static Analysis of Matching

Acceptable Programs. Before an external program can be accepted into a given net it has to be analysed with respect to an access policy defined by the accepting process. This ensures that the accepting process can control what access privileges it is willing to pass onto programs that may be unknown *a priori*. For an accepting process, $l ::^{e\delta}$ **accept**$(\delta').P$, willing to admit external programs, Q, that comply with policy δ' this check amounts to the following requirement on Q:

$$(\hat{T}, \Delta, \hat{\sigma}) \models^{\{l\}}_{\mathrm{P}} Q : (\!|\delta'|\!)^{\{l\}}, \bot$$

The check guarantees that a process Q, when evaluated at locality l, will only perform actions that do not violate the accepting policy, δ', as indicated by $(\!|\delta'|\!)^{\{l\}}, \bot$ on the right hand side of the colon. Here $\hat{T}$, Δ and $\hat{\sigma}$ should be considered "global constants" to be used for an entire execution of a net; this will be clarified in Theorem 1 below. Thus we may complete the semantics in Figure 4 by letting $\phi_{acc} = (\hat{T}, \Delta, \hat{\sigma}) \models^{\{l\}}_{\mathrm{P}} Q : (\!|\delta'|\!)^{\{l\}}, \bot$.

Analysis of the Running Example. For the running example, we have $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot$ for the following choice of $\hat{T}$, Δ and $\hat{\sigma}$:

$$\begin{array}{lll}
\hat{T} : l_U \mapsto \{\langle l_{C1}, \text{"The Hobbit"}\rangle\} & \hat{\sigma} : u \mapsto \{l_{C1}, l_{C2}\} & \Delta : l_U \mapsto \bot \\
\quad l_D \mapsto \{\langle \text{library}, l_{C1}\rangle, \langle \text{library}, l_{C2}\rangle\} & \quad \mathit{title} \mapsto \{\text{The Hobbit}\} & \quad l_D \mapsto \delta \\
\quad l_{C1} \mapsto \{\langle \text{J.R.R.Tolkien, The Hobbit}\rangle\} & \quad \mathit{source} \mapsto \{l_{C1}\} & \quad l_{C1} \mapsto \delta \\
\quad l_{C2} \mapsto \emptyset & \quad \mathit{data} \mapsto \{\text{The Hobbit}\} & \quad l_{C2} \mapsto \delta
\end{array}$$

Properties of the Analysis. Consistency of the analysis is formalised as a subject-reduction theorem.

Theorem 1 (Subject Reduction). *If* $L \triangleright N \longrightarrow_{\mathsf{off}} N'$ *and* $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \bot$*, then* $(\hat{T}, \Delta, \hat{\sigma}) \models_N N' : \bot$.

Proof. The proof is by induction on $L \triangleright N \longrightarrow_{\mathsf{off}} N'$, using a few auxiliary results:

- The analysis result is invariant under the structural congruence; that is, if $N \equiv N'$ then $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \varrho$ if and only if $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N' : \varrho$.
- The analysis of matching is correct; that is, if $\mathit{match}((\!|T|\!)_l, et) = \sigma$, $l \in \Lambda$, $et \in \hat{U}$, and $\hat{\sigma} \models^{\Lambda}_1 T : \hat{U} \triangleright \hat{W}$, then $et \in \hat{W}$ and $\forall u \in \mathsf{dom}(\sigma) : \sigma(u) \in \hat{\sigma}(u)$. ∎

Note that this result also holds with ϱ in place of $\bot$, but it is more instructive to consider executions where no security policy is violated; the result clearly does not hold if $\longrightarrow_{\mathsf{on}}$ is used (as any accepted process may violate the analysis and the security policy). Overall correctness of the analysis is formalised as an adequacy result.

Theorem 2 (Adequacy). *If* $L \triangleright N \longrightarrow_{\mathsf{off}} N'$ *and* $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \bot$*, then* $L \triangleright N \longrightarrow_{\mathsf{on}} N'$.

Proof. The proof is by induction on $L \triangleright N \longrightarrow_{\mathsf{off}} N'$, by inspecting Figures 5, 6, 7, and 8. ∎

More informally, we can show that if $L \triangleright N \longrightarrow_{\mathsf{off}} N'$ and $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \varrho$, then all offending actions performed are listed in ϱ. Finally, existence of best analysis estimates is formalised as a Moore-family result:

Theorem 3 (Moore Family). *For all nets* N*, the set* $\mathcal{Y}$ *of analysis estimates* $\{(\hat{T}, \Delta, \hat{\sigma}, \varrho) \mid (\hat{T}, \Delta, \hat{\sigma}) \models_N N : \varrho\}$ *is a* Moore Family*; i.e.,* $\forall Y \subseteq \mathcal{Y} : \sqcap Y \in \mathcal{Y}$.

Proof. The proof is by structural induction on N using that all constraints on $(\hat{T}, \Delta, \hat{\sigma}, \varrho)$ occur in positive positions only. ∎

Comparison with previous analyses of KLAIM. The analysis presented in this paper is an extension of a reworked version of the analysis specified in [6]. The main extension being an added Δ component to give a record of the policies imposed by the local **eval**'s. We have also reworked and rationalised the notation and introduced a number of auxiliary functions (most notably, $\langle\ \rangle$ and $[\ ]$) to increase readability of the analysis. Finally, we have added the Λ component (essentially allowing remotely evaluated processes to be analysed only once rather than at each receiving locality as in [6]). Among other things, this makes implementation easier.

4 A Static Type System

Typing approaches to KLAIM usually exploit dynamic checks; we now present a totally static type system whose design has been inspired by the Flow Logic developed in the previous section. We conclude this section by presenting the theoretical properties of the type system and the analysis of our running example.

Types and Auxiliary Functions. We can get rid of dynamic checks by following the philosophy underlying the Flow Logic approach. Indeed, it suffices to associate to every locality an upper bound of the tuples it can contain and a lower bound on its policy (like functions $\hat{T}$ and Δ did in Section 3); moreover, we should also provide an upper bound to the set of localities that can instantiate every variable. Thus, types for localities are pairs $\langle \mathcal{T}; \partial \rangle$, where $\mathcal{T} \subset_{\mathrm{fin}} \mathsf{Loc}^*$. Intuitively, if $\langle \mathcal{T}; \partial \rangle$ is the type of l, $\mathcal{T}$ is an upper bound on the tuples that l can contain and ∂ is a lower bound on l's policy. Types for input variables are, instead, just sets of localities; we can assign to u the type $\mathcal{T} \subset_{\mathrm{fin}} \mathsf{Loc}$, meaning that $\mathcal{T}$ are the localities that u can assume. A *typing environment* Γ assigns types to localities and variables.

Given a typing environment Γ, we now define some functions that will be used in the type system. First, we need to specify the values an identifier can assume. Thus, $\mathtt{val}_\Gamma(l) = \{l\}$ and $\mathtt{val}_\Gamma(u) = \Gamma(u)$; the definition of function $\mathtt{val}_\Gamma$ is extended to tuples component-wise. In the type system, we shall frequently look at the possible tuples a node can contain, at its policy or at the privileges it owns over the other nodes of the net. These pieces of information are easily accessible when the node is specified by a locality constant, thanks to the typing environment given. However, it can also happen in the typing phase to have nodes specified by variables (take, e.g., process $\mathbf{in}(!u)@l.\mathbf{eval}(Q : \delta)@u.P$, where Q must be typed at u). In this case, the information must be extracted from Γ as follows.

The tuples that can appear at a node identified by a variable are obtained by considering the tuples that can appear at every node whose locality is associated to the variable. However, from case to case, we need to know the tuples shared by all such nodes or all the possible tuples; accordingly, we combine the tuples contained at the different nodes by intersection or union. The following functions perform these tasks:

$$\Gamma\langle\ell\rangle = \bigcap_{l \in \mathtt{val}_\Gamma(\ell)} \pi_1(\Gamma(l)) \qquad \Gamma[\ell] = \bigcup_{l \in \mathtt{val}_\Gamma(\ell)} \pi_1(\Gamma(l))$$

To know the rights a policy grants over a node identified by a variable, we consider the intersection of all the privileges over the localities that the variable can assume:

$$\mathtt{Priv}_\Gamma(\partial, \ell) = \bigcap_{l \in \mathtt{val}_\Gamma(\ell)} \partial(l)$$

Similarly, the policy of a node identified by a variable is the greatest subset of access rights present in the policy of every locality that the variable can assume:

$$\mathtt{Pol}_\Gamma(\ell) = \bigsqcap_{l \in \mathtt{val}_\Gamma(\ell)} \pi_2(\Gamma(l))$$

where $\sqcap$ denotes the greatest lower bound.

In the typing rules, we shall need to evaluate localities and policies to replace occurrences of **self**. In both cases, we extend the evaluation function for localities and policies introduced when presenting the operational semantics to allow the subscript to also be a variable (in the case in which the node where the evaluation takes place is identified by a variable). This leads to notations $(\!|\ell'|\!)_\ell$ and $(\!|\delta|\!)^\Gamma_\ell$; for the latter, we have that $(\!|\delta|\!)^\Gamma_\ell(l)$ is $\delta(l)$, if $l \notin \mathtt{val}_\Gamma(\ell)$, and is $\delta(l) \cap \delta(\mathbf{self})$, otherwise.

Finally, given a typing environment Γ and a template T used by a process running at locality ℓ, we need to check that Γ provides the right information on the variables bound in T. Thus, we define the check of Γ with T at ℓ, written $check_\ell(\Gamma, T)$, as the judgement:

$$\forall i.\pi_i(T) = !u \Rightarrow$$
$$\pi_i(\{et \in \Gamma[\ell] \ : \ |et| = |T| \ \wedge \forall j \in \{1..|T|\}.\, \pi_j(T) = \ell' \Rightarrow \pi_j(et) \in \mathtt{val}_\Gamma(\ell')\}) \subseteq \Gamma(u)$$

In particular, every variable bound in T will be associated to all the possible localities that, at runtime, can be used to instantiate such a variable. The latters are obtained by taking all the possible tuples (of the same length as T and that can match against it) that can appear at ℓ and consider their i-th projection, for every i such that the i-th field of T is a variable.

$$\frac{\Gamma \vdash N_1 \quad \Gamma \vdash N_2}{\Gamma \vdash N_1 \parallel N_2} \qquad \frac{et \in \pi_1(\Gamma(l))}{\Gamma \vdash l :: \langle et \rangle} \qquad \frac{\pi_2(\Gamma(l)) \sqsubseteq e\delta \qquad \Gamma; e\delta \vdash_l P}{\Gamma \vdash l ::^{e\delta} P}$$

Fig. 9. Typing Nets

$$\frac{(\!|\ell'|\!)_\ell = \ell'' \qquad o \in \mathtt{Priv}_\Gamma(\partial, \ell'') \qquad \mathtt{val}_\Gamma((\!|t|\!)_\ell) \subseteq \Gamma\langle \ell'' \rangle \qquad \Gamma; \partial \vdash_\ell P}{\Gamma; \partial \vdash_\ell \mathbf{out}(t)@\ell'.P}$$

$$\frac{(\!|\ell'|\!)_\ell = \ell'' \qquad e \in \mathtt{Priv}_\Gamma(\partial, \ell'') \qquad (\!|\delta|\!)^\Gamma_\ell = \partial' \sqsubseteq \mathtt{Pol}_\Gamma(\ell'') \qquad \Gamma; \partial' \vdash_{\ell''} Q \qquad \Gamma; \partial \vdash_\ell P}{\Gamma; \partial \vdash_\ell \mathbf{eval}(Q : \delta)@\ell'.P}$$

$$\frac{(\!|\ell'|\!)_\ell = \ell'' \qquad i \in \mathtt{Priv}_\Gamma(\partial, \ell'') \qquad check_{\ell''}(\Gamma, (\!|T|\!)_\ell) \qquad \Gamma; \partial \vdash_\ell P}{\Gamma; \partial \vdash_\ell \mathbf{in}(T)@\ell'.P}$$

$$\frac{(\!|\ell'|\!)_\ell = \ell'' \qquad r \in \mathtt{Priv}_\Gamma(\partial, \ell'') \qquad check_{\ell''}(\Gamma, (\!|T|\!)_\ell) \qquad \Gamma; \partial \vdash_\ell P}{\Gamma; \partial \vdash_\ell \mathbf{read}(T)@\ell'.P}$$

$$\frac{a \in \mathtt{Priv}_\Gamma(\partial, \ell) \qquad (\!|\delta|\!)^\Gamma_\ell \sqsubseteq \partial \qquad \Gamma; \partial \vdash_\ell P}{\Gamma; \partial \vdash_\ell \mathbf{accept}(\delta).P} \qquad \frac{\Gamma; \partial \vdash_\ell P_1 \quad \Gamma; \partial \vdash_\ell P_2}{\Gamma; \partial \vdash_\ell P_1 \mid P_2} \qquad \frac{\Gamma; \partial \vdash_\ell P}{\Gamma; \partial \vdash_\ell *P}$$

Fig. 10. Typing Processes

Typing Rules. We are now ready to present the typing system. The typing rules for nets are in Figure 9; they define judgements of the form $\Gamma \vdash N$ that should be read as: "net N respects the constraints specified on its nodes by Γ". The rules are simple: to type a compound net we should type the components isolately; to type a located tuple, we must ensure that the tuple is allowed by Γ; to type a located process, we must ensure that the policy $e\delta$ conforms to the policy specified by Γ and that the process respects $e\delta$.

The typing rules for processes are in Figure 10 and define judgements of the form $\Gamma; \partial \vdash_\ell P$. Intuitively, such a judgement is needed to type under Γ a process P running at ℓ (where, by construction of the typing system, ℓ cannot be **self**) associated with policy ∂. The key rules are for action prefixes. In all cases, it is verified that the policy associated to the process provides a proper access right; moreover, to this aim, if the action can take place remotely, a preliminary evaluation of the locality target of the action is needed. For action **out**, the main thing to check is that the tuples that the action can produce can appear at every possible target locality (thus, we need here the intersection of all the possible tuple spaces, as calculated by $\Gamma\langle\rangle$); of course, we also have to check that the continuation is well-typed. For action **eval**, apart from checking that the continuation is well-typed, we have to check that the specified policy conforms to the policy associated to the target and, in this case, that the spawned process can run under the specified policy at the target locality. For actions **in** and **read**, we have to type the continuation in a typing environment obtained by extending the current environment with the possible values that variables bound in the template can assume. Finally, for

action **accept**, we only need to verify that the specified policy conforms to the policy of the hosting node and that the continuation is well-typed.

We can now complete the semantics in Figure 4 by using as ϕ in the rule for the **accept** action the judgement $\Gamma; \partial' \vdash_l Q$, where Γ is the typing environment used to type the net containing $l ::^{e\delta}$ **accept**$(\delta').P$ and $\partial' = (\!|\delta'|\!)_l$.

Soundness Results. A net N is *typeable* if there exists a Γ such that $\Gamma \vdash N$. We now prove that typeable nets are exactly the ones that can be accepted by the Flow Logic without errors; as a corollary of Theorems 1 and 2, this result trivially entails that also the type system enjoys subject reduction and adequacy.

Theorem 4. *N is typeable if and only if $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \bot$.*

Proof. (If) We first sketch how to prove that acceptable nets are typeable. To this aim, given a triple $(\hat{T}, \Delta, \hat{\sigma})$ and a net N such that $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \bot$, we define the typing environment Γ as follows:

$$\begin{array}{lll} \Gamma(u) & = & \hat{\sigma}(u) \qquad \text{for every } u \in \mathsf{LocVar} \\ \Gamma(l) & = & \langle \hat{T}(l); \partial_l \rangle \qquad \text{for every } l \in \mathsf{Loc}, \text{ where } \partial_l = \prod_{l ::^{e\delta} P \, in \, N} e\delta \end{array}$$

where "$l ::^{e\delta} P$ *in* N" means that $N \equiv l ::^{e\delta} P \parallel N'$, for some N'. Then, the proof works by induction on the length of the inference for $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \bot$, by exploiting two lemmata:

1. If $(\hat{T}, \Delta, \hat{\sigma}) \models_P^{\Lambda} P : \partial_1, \bot$ then $\Gamma; \partial_2 \vdash_\ell P$, whenever $\Lambda = \mathtt{val}_\Gamma(\ell)$ and $\partial_1 \sqsubseteq \partial_2$.
2. If $match((\!|T|\!)_l, et) = \sigma$, $l \in \Lambda$, $et \in \hat{U}$ and $\hat{\sigma} \models_1^{\Lambda} T : \hat{U} \triangleright \hat{W}$, then $et \in \hat{W}$ and $\sigma \sqsubseteq \hat{\sigma}$.

(Only if) We now sketch how to prove that typeable nets are acceptale. To this aim, given a typing environment Γ and a net N such that $\Gamma \vdash N$, we define the triple $(\hat{T}, \Delta, \hat{\sigma})$ as follows:

$$\begin{array}{lll} \hat{\sigma}(u) & = & \Gamma(u) \qquad \text{for every } u \in \mathsf{LocVar} \\ \hat{T}(l) & = & \pi_1(\Gamma(l)) \qquad \text{for every } l \in \mathsf{Loc} \end{array}$$

To define Δ, we first need to remove every occurrence of **self** occurring as target of actions in N as follows (we only give the non-homomorphic cases):

$$(\!|l ::^{e\delta} P|\!) = l ::^{e\delta} (\!|P|\!)_l \qquad (\!|\alpha.P|\!)_\ell = (\!|\alpha|\!)_\ell.(\!|P|\!)_\ell$$

$$(\!|\mathbf{out}(t)@\ell'|\!)_\ell = \mathbf{out}(t)@(\!|\ell'|\!)_\ell \qquad (\!|\mathbf{eval}(Q : \delta)@\ell'|\!)_\ell = \mathbf{eval}((\!|Q|\!)_{(\!|\ell'|\!)_\ell} : \delta)@(\!|\ell'|\!)_\ell$$

$$(\!|\mathbf{in}(T)@\ell'|\!)_\ell = \mathbf{in}(T)@(\!|\ell'|\!)_\ell \qquad (\!|\mathbf{read}(T)@\ell'|\!)_\ell = \mathbf{read}(T)@(\!|\ell'|\!)_\ell$$

Then, for every $l \in \mathsf{Loc}$, we let

$$\Delta(l) = \coprod_{\mathbf{eval}(P:\delta)@\ell \ in \ (\!|N|\!) \ : \ l \in \mathtt{val}_\Gamma(\ell)} (\!|\delta|\!)_l^\Gamma$$

The proof then works by induction on the length of the inference for $\Gamma \vdash N$, by exploiting two auxiliary lemmata:

1. If $\Gamma; \partial \vdash_\ell P$ and $\Lambda = \mathtt{val}_\Gamma(\ell)$, then $(\hat{T}, \Delta, \hat{\sigma}) \models_P^{\Lambda} P : \partial, \bot$ and, for every $\mathbf{eval}(Q : \delta)@\ell'$ in P, it holds that $(\!|\delta|\!)_l^\Gamma \sqsubseteq \pi_2(\Gamma(l))$, for every $l \in \mathtt{val}_\Gamma(\ell')$.
2. Let $l \in \Lambda$ and assume that, for every $et \in \hat{U} \cap \hat{W}$, it holds that $match((\!|T|\!)_l, et) = \sigma \sqsubseteq \hat{\sigma}$; then $\hat{\sigma} \models_1^{\Lambda} T : \hat{U} \triangleright \hat{W}$. ∎

Analysis of the Running Example. Thanks to the previous theorem, we know that the running example can be typed; by looking at the proof of Theorem 4 (that shows how to define a proper Γ out of $\hat{T}$, $\hat{\sigma}$ and the typed net N), we have that the following typing environment makes the running example typeable:

$$\Gamma(l_K) = \langle \hat{T}(l_K); e\delta_K \rangle \qquad \Gamma(x) = \hat{\sigma}(x)$$

for every $K \in \{U, D, C1, C2\}$ and $x \in \{u, title, source, data\}$.

Final Remarks. Notice that $\pi_2(\Gamma(l))$ and $\Delta(l)$ are both used to statically analyze migrations at l of a process labeled with a policy δ, but are defined and used in different ways. The former is a lower bound on the policy of the receiving node and, hence, δ (properly evaluated) must be lower than $\pi_2(\Gamma(l))$. The latter is an upper bound to the policy specified for the migrating process and, hence, $\Delta(l)$ must be greater than δ (properly evaluated). For this reason, $\pi_2(\Gamma(l))$ is defined as the greatest lower bound of the policies specified for nodes with address l; instead, $\Delta(l)$ is defined as the lowest upper bound of the policies specified for migrations at l. In this way, if we have two migrations at l (say, with policies δ_1 and δ_2) and the nodes $l ::^{e\delta_1} \cdots$ and $l ::^{e\delta_2} \cdots$, the type system checks that $\delta_i \sqsubseteq e\delta_1 \sqcap e\delta_2 = \pi_2(\Gamma(l))$, whereas the Flow Logic checks that $\Delta(l) = \delta_1 \sqcup \delta_2 \sqsubseteq e\delta_j$. These two checks are equivalent, in that they are both equivalent to $\delta_i \sqsubseteq e\delta_j$.

5 Conclusions and Further Work

We have considered a dialect of KLAIM, an experimental language designed for modeling and programming distributed systems with mobile components, and have presented an operational semantics for it that, by taking advantage of a reference monitor, permits controlling the kind of operations processes can perform at the different localities. We have then considered an alternative approach to access control based on Flow Logic that permits statically checking absence of access violations. Finally, we have reconsidered one of the type systems for access control previously developed that contained some dynamic checks, and, by exploiting concepts already used in the Flow Logic section, we have designed a fully static type system. To the best of our knowledge, this is the first completely static type system for controlling accesses in the context of a tuple space-based coordination language. We have also shown that both static approaches are sound with respect to the dynamic one based on reference monitor and provide the same analysis results.

We see this work just as an initial step towards understanding the relationships between static and dynamic approaches to access control and studying the relative merit of type systems and Flow Logic specifications (expanding on [7]). In future work, we want to investigate the impact of extending the analysis to a language with a primitive for dynamically creating new nodes with assigned policies (this is usually called **newloc** in the KLAIM setting). Indeed, the semantics treatment of such a primitive would require the policies of nodes to change dynamically. Clearly, making policies on nodes much more dynamic, would entail a number of differences in the static analysis, that was never conceived to cater for this possibility. We also want to study the relationships

between the global approach of type systems and Flow Logic and the more local one of the more traditional type systems that may contain dynamic components. Finally, we find it challenging to understand the relative expressive power of reference monitors and the static analysis approaches also in light of the considerations of [11], where it is claimed that the two approaches can capture different properties and are somehow incomparable. It would be interesting to understand what assumptions on the models are necessary to guarantee relative soundness.

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

References

1. A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
2. 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, May 1998.
3. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
4. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
5. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *Proc. of ICALP'03*, volume 2719 of *LNCS*, pages 119–132. Springer, 2003.
6. R.R. Hansen, C.W. Probst, and F. Nielson. Sandboxing in myKlaim. In *The First International Conference on Availability, Reliability and Security, ARES'06*, Vienna, Austria, April 2006. IEEE Computer Society.
7. F. Nielson and H. Riis Nielson. Types from control flow analysis. In *Program Analysis and Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2007.
8. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, Berlin, Germany, second edition, 2005.
9. F. Nielson, H. Seidl, and H. Riis Nielson. A succinct solver for alfp. *Nord. J. Comput.*, 9(4):335–372, 2002.
10. H. Riis Nielson and F. Nielson. Flow logics: a multi-paradigmatic approach to static analysis. In *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS no. 2566, pages 223–244. Springer-Verlag, 2002.
11. F.B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics - 10 Years Back 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101. Springer, 2001.
12. N. Izura Udzir, A.M. Wood, and J.L. Jacob. Coordination with multicapabilities. *Sci. Comput. Program.*, 64(2):205–222, 2007.