From Flow Logic to Static Type Systems for Coordination Languages[☆]

Rocco De Nicola[a,], Daniele Gorla[b], René Rydhof Hansen[c], Flemming Nielson[d], Hanne Riis Nielson[d], Christian W. Probst[d], Rosario Pugliese[a,*]

[a]*Dipartimento di Sistemi e Informatica, Università di Firenze*
[b]*Dipartimento di Informatica, Università di Roma "La Sapienza"*
[c]*Department of Computer Science, Aalborg University*
[d]*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 the insights from the Flow Logic approach can be used to construct a type system for *statically* ensuring secure access to tuple spaces and safe process migration for an extension of the language KLAIM.

Key words: Global computing, Coordination languages, Formal methods, Flow logic, Type systems

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

[☆]This work is partially based on two preliminary papers, [12] and [17]. The work has been partially supported by the EU project SENSORIA, IST-2005-016004.

[*]Corresponding author

Email addresses: rocco.denicola@unifi.it (Rocco De Nicola), gorla@di.uniroma1.it (Daniele Gorla), rrh@cs.aau.dk (René Rydhof Hansen), nielson@imm.dtu.dk (Flemming Nielson), riis@imm.dtu.dk (Hanne Riis Nielson), probst@imm.dtu.dk (Christian W. Probst), pugliese@dsi.unifi.it (Rosario Pugliese)

accomplish shared goals. These languages are often 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 designed by relying either on specifying the *lists* of 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 scale better when users are distributed across organizational boundaries and to be more appropriate for open distributed systems (see e.g. [37]), because capabilities can be distributed to the subjects and can be passed on. Moreover, different categories of capabilities need not be statically fixed.

Different techniques have been devised not only to specify but also to enforce access control (see e.g. [36]). The most traditional one is based on a *reference monitor* that dynamically intercepts each attempted access to any (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. To limit these drawbacks, many *static analysis* techniques [28] have been devised. These techniques originate from the work on compilers [1] where it is imperative that all relevant behaviour of systems be statically determined. The result of analyzing a program is an analysis estimate that gives a *global summary* of the properties of interest. However, these approaches often require knowledge of the full system which makes the analysis more difficult.

To overcome these limitations, hybrid approaches have been investigated that take advantage of both static and dynamic checks. This is the case in the capability-based type systems for KLAIM (*Kernel Language for Agents Interaction and Mobility*, [10]), a 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 the *generative communication* in Linda [14] with multiple shared tuple spaces.

In the capability-based type systems for KLAIM (see e.g. [11, 15]), 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 checks are 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 template. 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.

Of course, dynamic checks downgrade system performance and, thus, should be minimized. Therefore, in this paper we shall show how ideas from the Flow Logic approach [35] to static analysis can be used to enhance KLAIM's type systems with means for giving a global account of the behaviour of the system under analysis. Indeed, this seems necessary for dealing with the distributed nature of tuple spaces and furthermore, it allows us to develop a fully static type system.

The Flow Logic approach borrows from the type-based approach its compositionality in axiomatizing validity of analysis estimates for a given system. We shall formulate the correctness of our Flow Logic by a subject reduction result and we shall also establish a Moore family result showing that a best, i.e. most precise, analysis result exists; the actual computation of this usually requires global solution of a system of constraints [29]. Thanks to the tight correspondence between the Flow Logic and the type system, these properties hold for the type system as well.

As a further contribution of this paper, the version of the language KLAIM that we take into account is enhanced with a novel construct, named **accept**, that allows us to model truly open systems, i.e. systems where new code can be injected from the outside. In similar scenarios, a reference monitor semantics is usually used to ensure that the security of the overall system is not violated. One of the main results of our work is that the static analysis (and the equivalent static type system) provides a mechanism for checking the behaviour of the new code, thereby allowing us to dispense with the reference monitor when the code does not violate certain simple conditions.

The rest of the paper is structured as follows. In Section 2 we present the syntax and semantics of KLAIM extended with the **accept** construct; we temporarily dispense with a primitive for creating new localities – this will be rectified in Section 5. We also introduce a running example that will be used throughout the paper to illustrate how the calculus, the Flow Logic and the type system work. A Flow Logic for the language is developed and proved correct in Section 3 and it is used as inspiration to design the fully static type system presented in Section 4. Our major result, proved in Section 4, shows that the two analysis techniques are in accordance. In Section 5 we show how our techniques can be tailored to accommodate the addition of a primitive for creating new localities and briefly discuss alternative formulations. We conclude in Section 6 with some hints for future work and a discussion of related work.

NETS

N	::=	$l ::^{e\delta} P$	process
	|	$l :: \langle et \rangle$	located tuple
	|	$N_1 \,\|\| N_2$	net composition

PROCESSES

P	::=	**nil**	empty process
	|	$\alpha.P$	action prefixing
	|	$P_1 \mid P_2$	parallel composition
	|	$*P$	replication

ACTIONS

α	::=	**out**$(t)@\ell$	output
	|	**in**$(T)@\ell$	input
	|	**read**$(T)@\ell$	read
	|	**eval**$(P : \delta)@\ell$	migration
	|	**accept**(δ)	admission

LOCALITIES

ℓ	::=	l	locality constant
	|	**self**	self
	|	u	locality variable

TEMPLATES

T	::=	ℓ	locality
	|	$!u$	input variable
	|	ℓ, T	multiple fields
	|	$!u, T$	multiple fields

TUPLES

t	::=	ℓ	element
	|	ℓ, t	multiple elements

EVALUATED TUPLES

et	::=	l	evaluated element
	|	l, et	multiple eval. elem.

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

Figure 1: Syntax of KLAIM

2. An Extension of KLAIM

In this section we introduce syntax and operational semantics of the extension of the language KLAIM we consider in this paper. We also present a running example that will be used in the rest of the paper for illustration purposes.

2.1. 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 the syntax of the capability-based types is presented in the lower part.

A net consists of processes or tuples located at a locality l, or of a composition of two nets. Processes are built up from the special process **nil**, that does not perform any action (and whose tailing occurrences are 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** 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. These actions are all classical KLAIM actions whereas the action **accept** is new: it enables (selectively) processes coming from the environment to get into the system. Indeed, **accept**, first introduced in [17], makes the language more suitable to model *open* systems, where processes are not necessarily known at the outset and can unpredictably appear during a computation.

For communication, we distinguish between *tuples* and *evaluated tuples*. An evaluated tuple is a sequence of values, that in our case are elements of the set Loc of localities, and can be stored in tuple spaces. In contrast, tuples can contain variables and self-references, denoted by **self**, that allow programmers to write processes in a location-independent way. Tuples are used in processes to compose data to be communicated. We will use $\pi_i(t)$ to denote the i-th component of the tuple t.

For selectively accessing tuples in tuple spaces and, hence, dynamically retrieving information, processes use *templates* and a *matching* function (see Figure 2). Templates are similar to tuples, but can also contain *input variables*, denoted as $!u$, that are *bound* in the continuation process. This means that actions **in** and **read** are *binders* for input variables. A variable occurrence 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*[1]. 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*, c, to be indicators of the access operation, i.e., elements of the set Capabilities representing the **out**-, **in**-, **read**-, **eval**-, and **accept**-actions, respectively. We use C to denote a generic set of capabilities, i.e., a subset of the set Capabilities. Policies are represented as *capability lists*. Thus, a policy for some locality l_s maps a locality l_o to the set of capabilities with which the subject l_s can access the object 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, $e\delta$, associated to some locality, written $l ::^{e\delta} \ldots$, cannot.

The policies associated to nodes at the outset and the policies specified in the actions

[1] Although the security model we take into account in this paper exploits the power of capabilities only in a limited way (see, e.g., the semantics of the **newloc** action in Section 5), we retain this terminology for uniformity with other papers on KLAIM [15, 16] where more sophisticated security models are considered.

eval and **accept** must be explicitly defined by the programmer as an integral part of the specification.

2.2. 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 scattered on network nodes. The KLAIM net modeling the scenario includes the user process (located at l_U), a bookshop service process (located at l_B), and a data container (located at l_C). Moreover, in the example we assume that some sort of primitive data, like 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 interpreted as localities. The overall structure of the network is as follows:

$$l_U ::^{e\delta_U} P_U \quad \| \quad l_B ::^{e\delta_B} P_B \quad \| \quad l_C ::^{e\delta_C} \mathbf{nil} \quad \|$$
$$l_C :: \langle \text{J.R.R. Tolkien, The Hobbit} \rangle \quad \| \quad l_C :: \langle \text{J.R.R. Tolkien, The Lord of the Rings} \rangle$$

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

$$\begin{array}{lcl} P_U & = & \mathbf{eval}(P_1 : \delta)@l_B.\mathbf{in}(!data)@\mathbf{self}. < elaborate\ data > \\ P_1 & = & \mathbf{read}(\text{J.R.R. Tolkien}, !title)@l_C.\mathbf{out}(title)@l_U \\ P_B & = & \mathbf{accept}(\delta_a) \end{array}$$

The policies associate with the **eval** and the **accept** actions are

$$\delta = [l_U \mapsto \{o\}, l_C \mapsto \{r\}] \qquad \delta_a = [l_C \mapsto \{r, o\}]$$

and the policies of the localities are

$$e\delta_U = [l_U \mapsto \{i\},\ l_B \mapsto \{e\}] \qquad e\delta_B = [l_C \mapsto \{r, i, o\},\ l_U \mapsto \{o\}] \qquad e\delta_C = [\ ]$$

The intended workflow in the example is as follows. The user launches process P_1 at l_B (this is permitted because $e \in e\delta_U(l_B)$). The process starts by looking for a book written by Tolkien at the repository l_C, by reading a tuple with first component "J.R.R. Tolkien" (this is permitted because $r \in \delta(l_C)$). When such a book is found, the user is informed by receiving a datum in its tuple space containing the title (this is permitted because $o \in \delta(l_U)$). The user then starts working on it, for example by deciding whether to buy and read the book or look for another one. The bookshop has inserted in its policy $e\delta_B$ also the i and o capabilities for updating the repository, when a book is out of print or when one becomes available.

Notice the presence of action $\mathbf{accept}(\delta_a)$ at the bookshop. It allows entrance of code not present at the outset; however, the accepted code will only be allowed to perform a

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

Figure 2: The partial matching function

limited set of operations (delimited by δ_a). For example, an incoming process can read the repository and update it by adding new titles; however, it cannot remove existing titles[2].

The example is intentionally simplistic and several features have been omitted. We aim only at illustrating how the calculus, the Flow Logic and the type system work. The example is not meant to be a complete specification of a distributed system and therefore it does not include modeling of, e.g., scheduling of different processes and similar concepts.

2.3. Operational Semantics

The operational semantics is in the form of a reduction semantics. It uses the partial function *match* of Figure 2 when reading or inputting, for checking agreement between a template and an (evaluated) tuple. 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; in both cases, the rest of the tuples must match. The result of a successful match is a *substitution* function, σ, that is used to replace in the continuation process all the free occurrences of the template's input variables with the values that occurred at corresponding positions in the matched tuple[3]. In the sequel, to ensure that *match* returns a valid substitution function upon successful matching, we shall assume that templates T are *well-formed* in the sense that, for every given u, they do not contain both u and $!u$, and do not contain multiple occurrences of $!u$.

The reduction semantics operates on closed processes, i.e. processes without free variables, but it still needs to properly deal with **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 locality to be used instead

[2]Of course, the use of more fine-grained capabilities (like those in [16]) would simplify the specification of more realistic policies, prescribing, e.g., that the new agent can only add/remove certain kinds of tuples and/or must authenticate before performing its tasks.

[3]As usual, 'ϵ' denotes the empty function and *juxtaposition*, e.g., $\sigma_1\sigma_2$, denotes composition of functions with disjoint domains.

of **self** and have the same syntax.

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

$$(\!|\cdot|\!)_l : \mathsf{Policy} \to \mathsf{EvaluatedPolicy} \quad \text{given by} \quad (\!|\delta|\!)_l(l') = \begin{cases} \delta(l') & \text{if } l \neq l' \\ \delta(l) \cap \delta(\mathbf{self}) & \text{if } l = l' \end{cases}$$

The first function simply replaces **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 imposes both the policies of l and **self** whenever $l' = l$.

Figure 3 shows the semantics for our calculus – or, to be more precise, it defines two different semantics, one that checks the access control policies dynamically, and one that checks them statically. In both cases the transitions take the form

$$L \triangleright N \longrightarrow N'$$

where the set $L \subset \mathsf{Loc}$ keeps track of used localities and is exploited for testing localities' existence. Given a net N and an L as above, the configuration $L \triangleright N$ is *well-formed* if all the localities syntactically occurring in N (both in its tuples, processes and policies) are contained in L. The semantics is defined only for well-formed configurations.

The *reference monitor semantics* is obtained by taking $\mathsf{RM}[\phi]$ to be simply ϕ, thereby reflecting that the conditions are checked *dynamically*; in the following, we shall refer to this semantics by writing $L \triangleright N \longrightarrow_{\mathsf{on}} N'$. 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 permit output to the locality l'. In the reference monitor semantics of the **accept** action, we check that l is ready to accept a process from the environment by the condition $\mathsf{RM}[e\delta(l) \ni a]$, whereas the condition $\overline{\mathsf{RM}}[\cdots]$ is ignored by taking it to be universally true.

The alternative semantics will perform the checks *statically* and dispenses with the dynamic checks of the reference monitor. It is obtained from Table 3 by taking $\mathsf{RM}[\cdots]$ to be universally true and, in the rule for **accept**, by letting $\overline{\mathsf{RM}}[\phi_{acc}]$ be ϕ_{acc}. The static analysis/type system to be developed in the next sections will specify ϕ_{acc} in details; intuitively, it ensures that a new process is admitted only if it complies with the policy specified as argument of the **accept** action. We shall refer to this semantics by writing $L \triangleright N \longrightarrow_{\mathsf{off}} N'$.

Returning to the details of the rules of Table 3, we first observe that the **out** action takes an evaluated tuple and outputs it at the tuple space identified by ℓ. 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

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

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

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

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

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

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

Figure 3: Operational Semantics of KLAIM

the judgment for *match* previously defined to select a tuple matching against T from the tuple space at ℓ. As an effect of the **in** action, the matched tuple *et* 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 with the corresponding values in *et*. The **read** action acts as the **in** but leaves the matched tuple in place.

The **eval** action sends its argument Q for execution to the locality identified by ℓ. Notice that the receiving node has no control on the incoming code. Of course, we could introduce a sort of complementary **coeval** action for authorizing migrations (in the same spirit as co-capabilities in [21]) but, for the sake of simplicity, we prefer the present version of the language. The 'sandbox' policy under which Q will run results from evaluation of the policy specified in the argument of the action.

The **accept** action admits into a system new processes coming from the environment (i.e. processes that do not occur in the term representing the net under consideration) that will run under the 'sandbox' policy resulting from evaluation of the policy specified as argument of the action.

In the rules for **eval** and **accept**, we do not impose any condition on the sandbox policy with respect to other policies that might apply at the target locality; thus, it is expectable to have processes at the same locality but with different policies. Of course, it is possible to add a runtime check imposing that the sandbox policy is less permissive than the policy of the target node. In the rule for **accept**, this is simply done by adding the premise $e\delta' \sqsubseteq e\delta$,

$$N_1 \,||\, N_2 \;\equiv\; N_2 \,||\, N_1 \qquad (N_1 \,||\, N_2) \,||\, N_3 \;\equiv\; N_1 \,||\, (N_2 \,||\, N_3) \qquad l ::^{e\delta} *P \;\equiv\; l ::^{e\delta} P \,|\, *P$$

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

Figure 4: Structural Congruence

where '$\sqsubseteq$' is inclusion of partial functions (see Section 3.1 for a formal definition); in the rule for **eval**, this is done by checking the existence of a node at locality l' with some policy $e\delta''$ and by adding the premise $e\delta' \sqsubseteq e\delta''$. Although from a security point of view this modeling could seem more convincing, we prefer not to implement it because of two reasons: first, it requires more runtime checks, that downgrade system performance and are not necessary when working with nets that have already passed a static analysis phase (the only ones we are really interested in); second, the focus of this work is on the relationships between the Flow Logic and the type system, thus we prefer to keep the language under consideration as simple as possible. However, it has to be said that our choice leads to some nets that can safely run with the reference monitor on, but that cannot pass the static checks; we shall be back on this point in Section 3.9, after having presented the Flow Logic.

To conclude, the last two rules in Figure 3 are quite standard: the former allows a net to evolve whenever a subnet evolves, the latter assigns the same semantics to nets related by the structural congruence relation defined by the laws in Figure 4. In fact, as usual, reductions are given up-to a (quite standard) structural congruence. Its laws say that || is commutative and associative, that as many copies as needed can be spawned of a replicated process, that process **nil** can be absorbed/spawned, and that a parallel between co-located processes can be turned into a parallel between nodes (and viceversa, provided that the policies coincide). As a consequence, also | is commutative and associative, and has **nil** as identity element.

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. The analysis is specified using the *Flow Logic* approach to static analysis. A Flow Logic specification axiomatizes when an analysis estimate is acceptable for a program and relies on various algorithms for computing the analysis estimate – just like a type system specifies when a program is type correct and relies on type inference algorithms to actually construct the types. Thus, Flow Logic is a more specification oriented approach to program analysis than traditional methods, e.g., data flow analysis (see [28]).

Flow Logic specifications usually take the form of a number of judgments one for each syntactic category to be analyzed. The judgments specify when an analysis estimate is acceptable for a fragment of a term from the relevant syntactic category. In the rest of the section, we first introduce the abstract domains underlying the analysis, then define the judgments for actions, processes, nets, and matchings, finally present the theoretical properties of the analysis.

3.1. Analysis Domains

In our analysis we shall make use of 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 *might* at some point during the execution 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 during execution 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*; it summarizes all the concrete policies that a given locality might have at some point during the execution. 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 $l \in \mathrm{dom}(\partial)$.

- $\Delta \in \mathsf{Loc} \to \mathsf{AbstractPolicy}$ is a record of policies that may arise at localities during execution due to remotely executed processes. For every target locality of a remotely executed process (i.e., targets of an **eval**-action), it summarizes all the "sandbox" policies specified by the **eval**-action in question. This results in an over-approximation of the possible actions taken by processes remotely executed at a given locality.

- $\varrho \in \mathsf{Loc} \to \mathsf{Loc} \to \mathcal{P}(\mathsf{Capabilities})$ is a *record of potential violations of policies*. It records all the actions that *might* have been performed during the evolution of the net and where it was *not* possible to determine that the actions were 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. Note that this domain is equivalent to $\mathsf{Loc} \to \mathsf{AbstractPolicy}$; however, since policy violations are conceptually different from abstract policies, we retain the above definition.

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

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

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

Figure 5: Static Analysis of Actions

- $\Lambda \in \mathcal{P}(\mathsf{Loc})$ is a (usually nonempty) set of localities of interest at a given point. In general, we shall analyze processes at *sets* of localities (rather than at a single locality) in order to obtain a context insensitive analysis and to keep the complexity of the analysis low. A more precise analysis can be obtained by analyzing processes at single localities but it would require that some processes are analyzed more than once – indeed this approach is taken in [17].

3.2. Analysis of Actions

The judgment for an action α has the following form:

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

and is defined by the inference system of Figure 5 to be explained in more detail below. Intuitively, the above judgment reads: "$(\hat{T}, \Delta, \hat{\sigma})$ is an acceptable analysis estimate for the action α when occurring in the context Λ and it can only give rise to the policy violations recorded in ϱ and it can only impose the policy requirements ∂ on other localities". In other words: $\hat{T}$, Δ, and $\hat{\sigma}$ correctly capture the behaviour of the action α when executed in the context Λ and at the same time ∂ and ϱ provide records of the potential policies that α might impose on other localities and the potential policy violations that α might give rise to. As is usual in Flow Logic, we provide a componentwise definition.

Before explaining the rules of Table 5 we introduce some notation. We shall need to transform localities $\ell \in \mathsf{Loc} \cup \{\mathbf{self}\} \cup \mathsf{LocVar}$ into the set of localities that they denote; this is necessary both because we have locality variables and because we have the **self**

construct. For this we make use of the auxiliary function

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

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. As an example, in the rule for **out** we write $(\!|t|\!)^{\Lambda}_{\hat{\sigma}}$ for the set of potential tuples that could be output to one of the localities of $(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}$. It is easy to see that for evaluated tuples et, we have $(\!|et|\!)^{\Lambda}_{\hat{\sigma}} = \{et\}$.

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

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

Here X is the set of localities where the actions recorded in Y might have effect and Y is usually is a singleton set, namely the action taking place. 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. As an example, for the **out** action we will write $[(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \to \{o\}\,] \sqsubseteq \partial$ to record that at any of the localities of $(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}$ we might perform an **out** action and this must be recorded in the overall abstract policy ∂ of the particular occurrence of the action we are interested in. As another example, for the **accept** action we will have the somewhat simpler condition $[\Lambda \to \{a\}\,] \sqsubseteq \partial$ where we exploit that the process accepted from the environment will have to be executed at the current locality which will be one of the localities in Λ.

Since most of the rules need to take effect for any element in some set 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:

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

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

$$\Psi[X] \subseteq Z \iff \forall x \in X\colon \Psi(x) \subseteq Z$$
$$Z \subseteq \Psi\langle X\rangle \iff \forall x \in X\colon Z \subseteq \Psi(x)$$

As an example, in the rule for **out** 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.

The semantics of **in** and **read** make use of pattern matching and we also need to capture this in the analysis. We have developed an analysis for this making use of judgments of the form

$$\hat\sigma \models^\Lambda_1 T : \hat U \triangleright \hat W$$

The detailed definition is given later; at this stage it is sufficient to know that the judgment expresses that $\hat W$ contains the tuples of $\hat U$ that can be successfully matched against the template T in the context Λ, and that $\hat\sigma$ records the corresponding bindings to the variables of T. In the rules for **in** and **read** we make use of the premise $\hat\sigma \models_1^{(\!|\ell|\!)^\Lambda_{\hat\sigma}} T : \hat U \triangleright \hat W$ where $\hat U = \hat T[(\!|\ell|\!)^\Lambda_{\hat\sigma}]$. This expresses that the template T is matched against all the possible tuples of $\hat U$, which is constructed as the union of all the sets of tuples that the localities of ℓ might denote. In the analysis we are mainly interested in the bindings that the successful matches impose on $\hat\sigma$ so the actual set of successful matches $\hat W$ is not used in the analysis.

The next piece of notation needed in order to explain Table 5 is concerned with how to transform *concrete* policies into *abstract* policies. Here we make use of two functions; the first, denoted $(\!|\delta|\!)^\Lambda_\cup$, is used in conjunction with the ∂ component of the analysis to over-approximate the *potential* set of actions permitted by the concrete policy δ. It is defined by

$$(\!|\delta|\!)^\Lambda_\cup(\lambda) = \bigcup_{\lambda' \in \Lambda} (\!|\delta|\!)^{\{\lambda'\}}(\lambda)$$

$$(\!|\delta|\!)^{\{\lambda'\}}(\lambda) = \begin{cases} \delta(\lambda) & \text{if } \lambda \neq \lambda' \\ \delta(\lambda) \cap \delta(\mathbf{self}) & \text{if } \lambda = \lambda' \end{cases}$$

where the use of $(\!|\delta|\!)^{\{\lambda'\}}(\lambda)$ makes sure to include contributions from **self** of δ in the abstract policy. As an example, the rule for **accept** includes the premise $(\!|\delta|\!)^\Lambda_\cup \sqsubseteq \partial$ in order to ensure that the abstract policy ∂ records all the actions that a process accepted from the environment might perform. Similarly the rule for **eval** includes the premise $\forall \lambda \in (\!|\ell|\!)^\Lambda_{\hat\sigma} : (\!|\delta|\!)^\Lambda_\cup \sqsubseteq \Delta(\lambda)$ to ensure that the correct information is recorded in Δ for all the localities where a process might be remotely executed at.

The second function, denoted $(\!|\delta|\!)^\Lambda_\cap$, is used to compute the set of actions *definitely* permitted by a concrete policy. This results in an under-approximation that can be used to

remove definitely allowed actions from the error component, ϱ, and thereby improve the accuracy of the analysis. The function is defined by

$$(\!|\delta|\!)^{\Lambda}_{\cap}(\lambda) \quad = \quad \bigcap_{\lambda' \in \Lambda} (\!|\delta|\!)^{\{\lambda'\}}(\lambda)$$

where $(\!|\delta|\!)^{\{\lambda'\}}$ is as above.

The definitions of $(\!|\delta|\!)^{\Lambda}_{\cup}$ and $(\!|\delta|\!)^{\Lambda}_{\cap}$ are based on the observation that $(\!|\delta|\!)^{\{\lambda'\}}$ is the "precise" function that is needed when we know that the only possible value of **self** is the single value λ' (hence the superscript is $\{\lambda'\}$). We already observed that the set Λ of possible localities for **self** will in practice never be empty because we will always be at least at one place. Then the over-approximation is taken to be the pointwise union and the under-approximation is taken to be the pointwise intersection of the "precise" function.

The under-approximation is used in the rule for **eval** where one of the premises is $\partial' \setminus_{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}} (\!|\delta|\!)^{\Lambda}_{\cap} \sqsubseteq \varrho$. Here ∂' is the abstract policy obtained by analyzing the remotely executed process P and $(\!|\delta|\!)^{\Lambda}_{\cap}$ is a record of the access rights that we have specified for P. However, before explaining that in detail, we need to introduce an operation for "subtracting" two policies:

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

The subscript Λ is used to identify the localities where the subtraction should have effect. In the premise $\partial' \setminus_{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}} (\!|\delta|\!)^{\Lambda}_{\cap} \sqsubseteq \varrho$ discussed above we are only concerned about access violations at the localities where the new process may be spawned (that is, $(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}$) and we want the analysis to record in ϱ the violations that potentially might be problematic, that is, the difference between those recorded by the abstract policy obtained by analyzing the spawned process (that is, the over-approximation ∂') and those that definitely are non-problematic (that is, $(\!|\delta|\!)^{\Lambda}_{\cap}$). Since the latter is an under-approximation, the subtraction $\partial' \setminus_{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}} (\!|\delta|\!)^{\Lambda}_{\cap}$ will still be an over-approximation.

Having these notations in place, we can now return to the rules in Figure 5 on page 12. The rule for **out** evaluates the tuple t using $(\!|t|\!)^{\Lambda}_{\hat{\sigma}}$ to identify all possible tuples that could be output. Similarly, $(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}$ identifies all localities that could be the target of the action. The subset constraint $(\!|t|\!)^{\Lambda}_{\hat{\sigma}} \subseteq \hat{T}\langle (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \rangle$ ensures that all possible tuples are stored in the abstract tuple spaces of all possible target localities. Additionally, we record in the abstract policy ∂ that **out** could have been performed on all possible target localities.

The remaining rules are similar and we only point out novel features. In the rule for **in** we first need to pattern match the template T against all the possible tuples that could be input. The idea behind the analysis has been sketched above, pattern matching itself is discussed in Section 3.5. The analysis ensures that the environment $\hat{\sigma}$ correctly represents

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

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

Figure 6: Static Analysis of Processes

bindings of free variables in the template. The rule for the non-destructive **read** proceeds identically; this is because the analysis needs to be conservative. The rule for **accept** uses $(\!|\delta|\!)^{\Lambda}_{\cup} \sqsubseteq \partial$ to ensure that the abstract policy ∂ records all the actions that a process accepted from the environment might perform.

The rule for **eval** is the most complex one. To ensure that the analysis correctly captures all possibly executed actions, the process P is analyzed in all localities it possibly could be executed at. From the abstract policy ∂' obtained from analyzing the process P we remove all those privileges that have been definitely granted to P. The remaining elements are added to the error component. As for **accept**, we must also ensure that Δ records all the actions that process P might possibly perform, for all the localities where P might be remotely executed at.

3.3. Analysis of Processes

Let us now turn our attention to the analysis of processes. The judgment for the analysis of a process P has the form

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

and is defined by the inference system of Figure 6 to be explained shortly. The intention is that when true, the components $\hat{T}$, Δ, $\hat{\sigma}$, ∂ and ϱ correctly capture the behaviour of the process P (when located at one of the localities $\lambda \in \Lambda$). Any violation encountered during analysis of the process is recorded in ϱ, whereas ∂ approximates the actual policy employed by the process.

We now return to the rules of Figure 6. The rule for **nil** should be obvious. For processes in parallel we analyze both of them *using the same error component and abstract policy* as for their parallel composition. This is merely for simplicity: an alternative would be to use premises $(\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P_i : \partial_i, \varrho_i$ with different error components and abstract policies and then additionally require that $\partial_i \sqsubseteq \partial$ and $\varrho_i \sqsubseteq \varrho$ in order to conclude $(\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P_1 \,|\, P_2 : \partial, \varrho$. The same holds for the rule for action prefixing, where both the action and the remaining process are analyzed using the same components. Finally, we

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

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

Figure 7: Static Analysis of Nets

note that the analysis for replication only considers the process once as this is sufficient for collecting the relevant information.

3.4. Analysis of Nets

The judgment for a net N has the following form:

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

and intuitively it reads: "$(\hat{T},\Delta,\hat{\sigma})$ is an acceptable analysis estimate for the net N and it will only give rise to the policy violations recorded in ϱ". The judgment is defined by the rules of Table 7 that we now comment on.

When analysing a process P located at locality l with policy $e\delta$ we first analyse that process. Using the subtraction operator introduced earlier we ensure that all actions possibly performed by P but not allowed by the policy $e\delta$ are included in the error component. Finally, also the error component ϱ' obtained from P must be included in the overall error component ϱ. For an evaluated tuple located at l we record that the tuple is stored at the abstract tuple space $\hat{T}(l)$. Finally, parallel nets are analyzed individually using the same error component as explained above (viz. for parallel processes).

3.5. Analysis of Matching

In the analysis of the **in** and **read** actions in Figure 5, we made use of a judgment for analyzing pattern matching that has the following general form:

$$\hat{\sigma} \models_i^{\Lambda} T : \hat{U} \triangleright \hat{W}$$

It expresses that matching should start at position i in the template T, that $\hat{U}$ contains the set of tuples that we are matching against, that $\hat{W}$ contains the tuples from $\hat{U}$ that successfully match T from position i and onwards, and finally that $\hat{\sigma}$ records the appropriate bindings that need to be performed. In the rules of Figure 8, recall that $\pi_i(et)$ denotes the i-th component of the tuple et; $\pi_i(\hat{V})$ is its componentwise extension to sets of tuples.

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

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

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

Figure 8: Static Analysis of Matching

Pattern matching is triggered by the input rules in Figure 5 by invoking it with $i = 1$ and with $\hat{U}$ being initialized to the abstract tuple space from which the input action is reading. The rules in Figure 8 then traverse the template from left to right to construct a set of tuples that could match. After this, each element of the template is inspected again, and tuples that do not match are deleted.

We first look at the traversal of templates from left to right (last two rules of Figure 8). If the current template element is an input variable, all sufficiently long tuples from $\hat{U}$ are selected. If the element is a locality constant ℓ then only tuples in $\hat{U}$ with ℓ at position i are selected. The resulting set $\hat{V}$ is then forwarded to the analysis of the rest of the template for $i + 1$.

If the end of the template is reached, the first two rules of Figure 8 apply. Both rules apply the same constraints as their siblings just discussed. Additionally they select only those tuples that have the correct length; since we now are at the last element of the template, we know that the index i is the length of tuples we can match. Furthermore in the case of an input variable u we ensure that the i-th component of the remaining tuples is added to the abstract environment of u. Once these rules end the left-right traversal of the template, this is also ensured for all input variables encountered before (last rule of Figure 8).

3.6. Acceptable Terms

If we for a moment ignore the **accept** action, we would be ready to establish the semantic correctness of our analysis. This would be expressed as a *subject reduction result*:

$$\text{If } L \triangleright N \longrightarrow_{\mathsf{off}} N' \text{ and } (\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot, \text{ then } (\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N' : \bot.$$

Here $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot$ expresses that the net N is *policy conformant* and the above result states that this property is preserved by the semantics even when dispensing with

the reference monitor. Also, we would be able to show that the semantics without the reference monitor does not allow more transitions than the one with the reference monitor as expressed by the *adequacy result*:

If $L \triangleright N \longrightarrow_{\mathsf{off}} N'$ and $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot$, then $L \triangleright N \longrightarrow_{\mathsf{on}} N'$.

However, our calculus does include the **accept** action; so, we are not ready to prove the above results for the full calculus. We first need to finalize the semantics presented in Table 3 and, in particular, to define the premise $\overline{\mathsf{RM}}[\phi_{acc}]$ of the rule for **accept**. The idea is that an external process can be accepted into a given net if it can be analyzed 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 an external process Q that complies with policy δ', this check amounts to the following requirement on Q:

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

The check guarantees that process Q, when executed 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 3.1 below. Thus, we may complete the semantics in Figure 3 by letting

$$\phi_{acc} = (\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} Q : (\!|\delta'|\!)^{\{l\}}, \bot$$

3.7. 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 $\hat{\sigma}$:

$$\begin{array}{lllll}
\hat{T}: & l_U & \mapsto & \{\langle\text{The Hobbit}\rangle, \langle\text{The Lord of the Rings}\rangle, \langle\text{The Silmarillion}\rangle\} \\
& l_B & \mapsto & \{\} \\
& l_C & \mapsto & \{\langle\text{J.R.R. Tolkien, The Hobbit}\rangle, \langle\text{J.R.R. Tolkien, The Lord of the Rings}\rangle, \\
& & & \quad \langle\text{J.R.R. Tolkien, The Silmarillion}\rangle\} \\
\hat{\sigma}: & \mathit{title} & \mapsto & \{\text{The Hobbit, The Lord of the Rings, The Silmarillion}\} \\
& \mathit{data} & \mapsto & \{\text{The Hobbit, The Lord of the Rings, The Silmarillion}\} \\
\Delta: & l_U & \mapsto & \bot \\
& l_B & \mapsto & \delta \\
& l_C & \mapsto & \bot
\end{array}$$

This is an acceptable analysis result for the net although it is not the best (or least) solution. Consider now the two processes

$$\begin{array}{lcl}
Q_1 & = & \textbf{out}(\text{J.R.R. Tolkien, The Silmarillion})@l_C \\
Q_2 & = & \textbf{in}(\text{J.R.R. Tolkien, The Hobbit})@l_C
\end{array}$$

Here $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l_B\}} Q_1 : \delta_a, \bot$ because $[l_C \mapsto \{o\}] \sqsubseteq \delta_a$ and this means that Q_1 can be accepted into the net. On the other hand, it is *not* the case that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l_B\}} Q_2 : \delta_a, \bot$ since $[l_C \mapsto \{i\}] \not\sqsubseteq \delta_a$ and this means that Q_2 will not be accepted into the net.

3.8. Properties of the Analysis

We are now ready to prove the results envisioned above. As already mentioned the overall correctness of the analysis is formalized as a subject-reduction and an adequacy theorem.

Theorem 3.1 (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:*
  If $N \equiv N'$ *then* $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \varrho$ *if and only if* $(\hat{T}, \Delta, \hat{\sigma}) \models_N N' : \varrho$.
  The proof is by induction on the proof tree establishing $N \equiv N'$ and is standard.

- *The analysis of matching is correct:*
  If $match(\langle\!|T|\!\rangle_l, et) = \sigma$, $l \in \Lambda$, $et \in \hat{U}$, *and* $\hat{\sigma} \models_1^{\Lambda} T : \hat{U} \triangleright \hat{W}$, *then* $et \in \hat{W}$ *and* $\forall u \in \mathsf{dom}(\sigma) : \sigma(u) \in \hat{\sigma}(u)$.
  The proof is by structural induction on the template T and is standard.

- *The analysis result is stable under substitution:*
  If $(\hat{T}, \Delta, \hat{\sigma}) \models_P^{\Lambda} P : \varrho$ *and* $\lambda \in \hat{\sigma}(u)$ *then* $(\hat{T}, \Delta, \hat{\sigma}) \models_P^{\Lambda} P[\lambda/u] : \varrho$.
  If $(\hat{T}, \Delta, \hat{\sigma}) \models_A^{\Lambda} \alpha : \partial, \varrho$ *and* $\lambda \in \hat{\sigma}(u)$ *then* $(\hat{T}, \Delta, \hat{\sigma}) \models_A^{\Lambda} \alpha[\lambda/u] : \partial, \varrho$.
  If $\hat{\sigma} \models_i^{\Lambda} T : \hat{U} \triangleright \hat{W}$ *and* $\lambda \in \hat{\sigma}(u)$ *then* $\hat{\sigma} \models_i^{\Lambda} T[\lambda/u] : \hat{U} \triangleright \hat{W}$.
  The proof is a standard proof by mutual structural induction (mutual structural induction is used because actions may occur inside processes as well as processes inside actions).

The proof of the main theorem is then fairly standard and we only show how to prove subject reduction for the **in** and **eval** actions.

Assume that $N = l ::^{e\delta} \mathbf{in}(T)@\ell.P \,\|\, l' :: \langle et \rangle$, $N' = l ::^{e\delta} P\sigma$, $match(\langle\!|T|\!\rangle_l, et) = \sigma$, $\langle\!|\ell|\!\rangle_l = l'$ such that $L \triangleright N \longrightarrow_{\mathsf{off}} N'$, and further assume that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} l ::^{e\delta} \mathbf{in}(T)@\ell.P \,\|\, l' :: \langle et \rangle : \bot$. By the premises of the rules for action prefixing and action **in**, we have that (for some ∂')

1. $match(\langle\!|T|\!\rangle_l, et) = \sigma$
2. $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} \mathbf{in}(T)@\ell : \partial', \bot$

3. $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} P : \partial', \bot$
4. $\partial' \setminus_{\{l\}} e\delta \sqsubseteq \bot$
5. $\Delta(l) \setminus_{\{l\}} e\delta \sqsubseteq \bot$

We have that $(\!|\ell|\!)_{\hat{\sigma}}^{\{l\}} = \{l'\}$ and we then get that $\hat{\sigma} \models_1^{(\!|\ell|\!)_{\hat{\sigma}}^{\{l\}}} T : \hat{T}[(\!|\ell|\!)_{\hat{\sigma}}^{\{l\}}] \triangleright \hat{W}$ using points (1) and (2); using next the auxiliary results stated above we have $\forall u \in \mathsf{dom}(\sigma) : \sigma(u) \in \hat{\sigma}(u)$. From point (3) and the auxiliary results stated above we get $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} P\sigma : \partial', \bot$. Combined with points (4), and (5) we arrive at $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} l ::^{e\delta} P\sigma : \bot$ which concludes the case.

Assume that $N = l ::^{e\delta} \mathbf{eval}(Q : \delta')@\ell.P$, $N' = l ::^{e\delta} P \,\|\, l' ::^{e\delta'} Q$, $(\!|\ell|\!)_l = l'$ such that $L \triangleright N \longrightarrow_{\mathsf{off}} N'$, and further assume that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} l ::^{e\delta} \mathbf{eval}(Q : \delta')@\ell.P : \bot$; by the premises of the semantic rule and analysis clause for **eval** we have that (for some ∂')

1. $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{A}}^{\{l\}} \mathbf{eval}(Q : \delta')@\ell : \partial', \bot$
2. $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} P : \partial', \bot$
3. $\partial' \setminus_{\{l\}} e\delta \sqsubseteq \bot$
4. $\Delta(l) \setminus_{\{l\}} e\delta \sqsubseteq \bot$

From points (2), (3), and (4) we get that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} l ::^{e\delta} P : \bot$ and from point (1) we have that (for some ∂'')

5. $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{(\!|\ell|\!)_{\hat{\sigma}}^{\{l\}}} Q : \partial'', \bot$
6. $\partial'' \setminus_{(\!|\ell|\!)_{\hat{\sigma}}^{\{l\}}} (\!|\delta'|\!)_{\cap}^{\{l\}} \sqsubseteq \bot$
7. $\forall \lambda \in (\!|\ell|\!)_{\hat{\sigma}}^{\{l\}} : (\!|\delta'|\!)_{\cup}^{\{l\}} \sqsubseteq \Delta(l)$

Point (5) gives us that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} Q : \partial'', \bot$ since $l' = (\!|\ell|\!)_l \in (\!|\ell|\!)_{\hat{\sigma}}^{\{l\}}$. We further have that $e\delta' = (\!|e\delta'|\!)_{\cap}^{\{l\}}$ and thus $\partial' \setminus_{\{l'\}} e\delta' \sqsubseteq \bot$. From point (7) above we get that $e\delta' \sqsubseteq \Delta(l')$. Combining all of the above we now have $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} l ::^{e\delta} P \,\|\, l' ::^{e\delta'} Q : \bot$ which concludes the case. ∎

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 $\overline{\mathsf{RM}}[\cdots]$ then equals true and any accepted process may violate not only the analysis but also the security policy).

Theorem 3.2 (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 7, 6, 5, and 8. It is straightforward and we only consider two cases.

We first consider the case of input. Suppose that

$$L \triangleright l ::^{e\delta} \mathbf{in}(T)@\ell.P \,\|\, l' :: \langle et \rangle \longrightarrow_{\mathsf{off}} l ::^{e\delta} P\sigma$$

because $(\!|\ell|\!)_l = l'$ and $match((\!|T|\!)_l, et) = \sigma$. To show that

$$L \triangleright l ::^{e\delta} \mathbf{in}(T)@\ell.P \,\|\, l' :: \langle et \rangle \longrightarrow_{\mathsf{on}} l ::^{e\delta} P\sigma$$

it suffices to additionally show that $e\delta(l') \ni i$. It is immediate from the assumption of Theorem 3.2 that

$$(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} \mathbf{in}(T)@\ell.P : \partial, \bot$$
$$\text{with } \partial \setminus_{\{l\}} e\delta \sqsubseteq \bot$$

Since Figure 5 ensures that $[\{l\} \to \{i\}\,] \sqsubseteq \partial$ this gives $e\delta(l') \ni i$.

Next we consider the case of accept. Suppose that

$$L \triangleright l ::^{e\delta} \mathbf{accept}(\delta').P \longrightarrow_{\mathsf{off}} l ::^{e\delta} P \,\|\, l ::^{e\delta'} Q$$

because $(\!|\delta'|\!)_l = e\delta'$ and ϕ_{acc}. To show that

$$L \triangleright l ::^{e\delta} \mathbf{accept}(\delta').P \longrightarrow_{\mathsf{on}} l ::^{e\delta} P \,\|\, l ::^{e\delta'} Q$$

it suffices to additionally show that $e\delta(l) \ni a$. It is immediate from the assumption of Theorem 3.2 that

$$(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} l ::^{e\delta} \mathbf{accept}(\delta').P : \partial, \bot$$
$$\text{with } \partial \setminus_{\{l\}} e\delta \sqsubseteq \bot$$

Since Figure 5 ensures that $[\{l\} \to \{a\}\,] \sqsubseteq \partial$ this gives $e\delta(l') \ni a$.

This finishes the proof. ∎

In fact, it can be shown that, if $L \triangleright N \longrightarrow_{\mathsf{off}} N'$ and $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \varrho$, then all offending actions performed are listed in ϱ.

Finally, the existence of best analysis estimates is formalized as a Moore-family (or model intersection) property. As a corollary we get that for all nets N there exist $\hat{T}$, Δ, $\hat{\sigma}$ and ϱ such that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \varrho$. Thus the analysis in itself does not impose any of the limitations or policies of the reference monitor. But surely only some nets can be analysed with $\varrho = \bot$ and in that case the results of above theorems apply.

Theorem 3.3 (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., $\forall Y \subseteq \mathcal{Y} : \sqcap Y \in \mathcal{Y}$ where $\sqcap$ is the greatest lower bound operation.*

Proof: We prove the result by mutual structural induction using also the following results.

- For all processes P, the set of analysis estimates $\{(\hat{T}, \Delta, \hat{\sigma}, \Lambda, \partial, \varrho) \mid (\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P : \partial, \varrho\}$ is a Moore Family.
- For all actions α, the set of analysis estimates $\{(\hat{T}, \Delta, \hat{\sigma}, \Lambda, \partial, \varrho) \mid (\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{A}} \alpha : \partial, \varrho\}$ is a Moore Family.
- For all templates T and indices i, the set of analysis estimates $\{(\hat{\sigma}, \Lambda, \hat{U}, \hat{W}) \mid \hat{\sigma} \models^{\Lambda}_{i} T : \hat{U} \triangleright \hat{W}\}$ is a Moore Family.

The proof of the main result is standard. Intuitively it uses that all "constraints" on the analysis information occur in "positive" positions only. It is by mutual structural induction because actions may occur in processes as well as processes in actions. We only illustrate the case of input.

Consider a family J of indices ranged over by j. Suppose that

$$\forall j \in J : (\hat{T}_j, \Delta_j, \hat{\sigma}_j) \models_A^{\Lambda_j} \mathbf{in}(T)@\ell : \partial_j, \varrho_j$$

By Figure 5 we have

$$\begin{array}{l} \forall j \in J : \hat{\sigma}_j \models_1^{(\!|\ell|\!)^{\Lambda_j}_{\hat{\sigma}_j}} T : \hat{T}_j[(\!|\ell|\!)^{\Lambda_j}_{\hat{\sigma}_j}] \rhd \hat{W}_j \\ \forall j \in J : [(\!|\ell|\!)^{\Lambda_j}_{\hat{\sigma}_j} \to \{i\}\,] \sqsubseteq \partial_j \end{array}$$

Next write

$$(\hat{T}_\star, \Delta_\star, \hat{\sigma}_\star, \Lambda_\star, \partial_\star, \varrho_\star) = \sqcap_j(\hat{T}_j, \Delta_j, \hat{\sigma}_j, \Lambda_j, \partial_j, \varrho_j)$$

From the induction hypothesis we have

$$\hat{\sigma}_\star \models_1^{(\!|\ell|\!)^{\Lambda_\star}_{\hat{\sigma}_\star}} T : \hat{T}_\star[(\!|\ell|\!)^{\Lambda_\star}_{\hat{\sigma}_\star}] \rhd \hat{W}_\star$$

since $\sqcap_j (\!|\ell|\!)^{\Lambda_j}_{\hat{\sigma}_j} = (\!|\ell|\!)^{\Lambda_\star}_{\hat{\sigma}_\star}$ and

$$\begin{array}{rcl} \sqcap_j \hat{T}_j[(\!|\ell|\!)^{\Lambda_j}_{\hat{\sigma}_j}] & = & \sqcap_j \{z \mid \exists x \in (\!|\ell|\!)^{\Lambda_j}_{\hat{\sigma}_j} : z \in \hat{T}_j(x)\} \\ & = & \{z \mid \exists x \in (\!|\ell|\!)^{\Lambda_\star}_{\hat{\sigma}_\star} : z \in \hat{T}_\star(x)\} \\ & = & \hat{T}_\star[(\!|\ell|\!)^{\Lambda_\star}_{\hat{\sigma}_\star}] \end{array}$$

Next, $\forall j \in J : [(\!|\ell|\!)^{\Lambda_\star}_{\hat{\sigma}_\star} \to \{i\}\,] \sqsubseteq \partial_j$ and hence $[(\!|\ell|\!)^{\Lambda_\star}_{\hat{\sigma}_\star} \to \{i\}\,] \sqsubseteq \partial_\star$. This suffices for showing

$$(\hat{T}_\star, \Delta_\star, \hat{\sigma}_\star) \models_A^{\Lambda_\star} \mathbf{in}(T)@\ell : \partial_\star, \varrho_\star$$

as desired. ∎

The Moore family result ensures that a best, or least, analysis result can be found but it does not give a constructive algorithm for finding the analysis result. To do so the idea is to develop an algorithm converting the clauses into constraints and in particular Alternation-free Least Fixed Point Logic [29] has proved very useful for expressing these constraints as it is the basis for obtaining efficient implementations using for example the Succinct Solver [29, 27].

3.9. Final Remarks

The analysis presented in this paper is an extension of a reworked version of the analysis specified in [17], 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 rationalized 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 executed processes to be analyzed only once rather than at each receiving locality as in [17]). This reduces the computational cost of computing the analysis result as it will only be necessary to analyse each fragment of a term once.

As we have already mentioned, the static analysis is a sound but not complete technique, as expectable. Indeed, there are examples of nets that can safely run with the reference monitor on, but that are not acceptable. This may happen, for example, when the policy associated to an **eval**-action is more permissive than the local policy of the receiving node. As a simple example, consider the net

$$l ::^{e\delta} \mathbf{eval}(P : \delta)@l' \quad \| \quad l' ::^{e\delta'} \mathbf{nil}$$

where $e\delta = [l' \mapsto \{e\}]$, $\delta = [l' \mapsto \{e, r\}]$ and $e\delta' = [l' \mapsto \{r\}]$. This net is not policy conformant – indeed it is analysable with $[l' \mapsto \{e\}]$ as the potential policy violations. To see this, note that the analysis component Δ must satisfy $\{e, r\} \subseteq \Delta(l')(l')$ (see the third premise of the last rule in Figure 5) and therefore the policy violations ϱ of the overall net will have to satisfy $\Delta(l') \setminus_{l'} e\delta' \sqsubseteq \varrho$ (see the third premise of the first rule of Figure 7). Since $e\delta'$ maps l' to $\{r\}$ it cannot be the case that $\varrho = \emptyset$. Nevertheless, in one reduction step the net reduces to

$$l ::^{e\delta} \mathbf{nil} \quad \| \quad l' ::^{\delta} P \quad \| \quad l' ::^{e\delta'} \mathbf{nil}$$

that can go on reducing also with the reference monitor on, assuming that P only performs **read**- and **eval**-actions over l' (by the way, the net so obtained will be policy conformant).

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.

4.1. 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 (like function $\hat{T}$ in Section 3) and a lower bound on its policy; moreover, we should also provide an upper bound to the set of localities that each variable can assume (like function $\hat{\sigma}$). Thus, types for localities are pairs $\langle \mathcal{T}; \partial \rangle$, where $\mathcal{T} \subset_{\text{fin}} \mathsf{Loc}^*$. 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 $\mathcal{T} \subset_{\text{fin}} \mathsf{Loc}$, 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 definitely 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 intersection of all the policies 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 execution takes place is identified by a variable). This leads to notations $(\!|\ell'|\!)_\ell$ and $(\!|\delta|\!)^\Gamma_\ell$: for the former, we have that $(\!|\ell'|\!)_\ell$ is ℓ', if $\ell' \neq$ **self**, and is ℓ otherwise; for the latter, we have that $(\!|\delta|\!)^\Gamma_\ell(l)$ is $\delta(l)$, if $l \notin \mathtt{val}_\Gamma(\ell)$, and is $\delta(l) \cap \delta($**self**$)$, otherwise.

Finally, given a typing environment Γ and a template T used by a process for matching tuples located at locality ℓ, we need to check that Γ provides an upper bound on the localities that variables bound in T can assume. This is needed to ensure that function $\mathtt{val}_\Gamma$ (and, consequently, functions $\mathtt{Priv}_\Gamma$, $\mathtt{Pol}_\Gamma$, $\Gamma\langle\cdot\rangle$ and $\Gamma[\cdot]$) correctly overapproximates the values (the privileges, the policy and the tuples, respectively) of every variable. Thus, we define the check of Γ with T at ℓ, written $check_\ell(\Gamma, T)$, as the judgment:

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

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. For every i such that the i-th field of T is a variable, these localities are obtained by taking all the possible tuples of the same length as T that can match against it and can appear at ℓ, and considering their i-th projection.

4.2. Typing Rules

We are now ready to present the typing system. The typing rules for processes are in Figure 9 and define judgments of the form $\Gamma; \partial \vdash_\ell P$. Intuitively, such a judgment 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; to this aim, if the action can take place remotely, a preliminary evaluation of the locality target of the action is needed. Moreover, it is also checked that the continuation is well-typed. There are then some other specific checks that depend on the action. For action **out**, it is checked that the tuples that the action can produce can appear at every possible target locality; thus, it is used here the intersection of all the possible tuple spaces, as calculated by $\Gamma\langle\rangle$. For action **eval**, it is checked 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 'sandbox' policy at the target locality. For actions **in** and **read**, it is checked that Γ provides the right

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

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

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

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

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

Figure 9: Typing Processes

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

Figure 10: Typing Nets

information on the variables bound in T. Finally, for action **accept**, it is checked that the specified policy conforms to the policy of the hosting node.

The typing rules for nets are in Figure 10; they define judgments 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 individually; 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, when located at l, the process respects $e\delta$.

We can now complete the semantics in Figure 3 by using as ϕ in the rule for the action **accept** the judgment $\Gamma; \partial' \vdash_l Q$, where Γ is the typing environment used to type the net containing $l ::^{e\delta}$ **accept**$(\delta').P$ and $\partial' = (\!|\delta'|\!)_l$.

4.3. 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 (in Section 3.6 such nets were called *policy conformant*); as a corollary of Theorems 3.1 and 3.2, this result trivially entails that also the type system enjoys subject reduction and adequacy.

Theorem 4.1 (Accordance of the analyses). *N is typeable if and only if it is policy conformant.*

To prove the theorem, we start by listing some preliminary results on some auxiliary functions of the Flow Logic and of the type system, whose proof is easily derivable from the corresponding definitions.

Proposition 4.2. *Let $\Lambda = \mathtt{val}_\Gamma(\ell)$, $\hat{T}(l) = \pi_1(\Gamma(l))$ for every l and $\hat{\sigma}(u) = \Gamma(u)$ for every u; then,*

1. $(\!|\ell'|\!)^\Lambda_{\hat{\sigma}} = \mathtt{val}_\Gamma((\!|\ell'|\!)_\ell)$;
2. $\hat{T}\langle(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}\rangle = \Gamma\langle(\!|\ell'|\!)_\ell\rangle$ *and* $\hat{T}[(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}] = \Gamma[(\!|\ell'|\!)_\ell]$;
3. $(\!|\delta|\!)^\Lambda = (\!|\delta|\!)^\Gamma_\ell$.

We now show that the analysis of matching in Figure 8 is correct; this will be needed here to prove that function $check_\ell(\Gamma, T)$ holds true whenever a read/input with target ℓ and template T has passed the static analysis. Also in this case, the proof easily follows from the definitions.

Proposition 4.3.

1. *If $match((\!|T|\!)_l, et) = \sigma$, $l \in \Lambda$, $et \in \hat{U}$ and $\hat{\sigma} \models^\Lambda_1 T : \hat{U} \triangleright \hat{W}$, then $et \in \hat{W}$ and $\sigma \sqsubseteq \hat{\sigma}$.*
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, $\hat{\sigma} \models^\Lambda_1 T : \hat{U} \triangleright \hat{W}$.*

We can now prove that policy conformant processes and nets are typeable; this suffices to prove the "if" part of Theorem 4.1. To this aim, given a triple $(\hat{T}, \Delta, \hat{\sigma})$ and a net N such that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot$, we define the typing environment Γ as follows:

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

where "$l ::^{e\delta} P$ *in* N" means that $N \equiv l ::^{e\delta} P \parallel N'$, for some N'.

Lemma 4.4. *If $(\hat{T}, \Delta, \hat{\sigma}) \models^\Lambda_P P : \partial_1, \bot$ then $\Gamma; \partial_2 \vdash_\ell P$, whenever $\Lambda = \mathtt{val}_\Gamma(\ell)$ and $\partial_1 \sqsubseteq \partial_2$.*

Proof: The proof is by induction on the structure of P. The base step is trivial; for the inductive step, we only give the most complex cases.

Assume that $P = \mathbf{eval}(Q : \delta)@\ell'.P'$; by the premises of the rules for action prefixing and action **eval**, we have that

1. $(\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P' : \partial_1, \bot$
2. $(\hat{T}, \Delta, \hat{\sigma}) \models^{(\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}}}_{\mathrm{P}} Q : \partial, \bot$
3. $\forall \lambda \in (\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}} : (\!|\delta|\!)^{\Lambda} \sqsubseteq \Delta(\lambda)$
4. $\partial \setminus_{(\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}}} (\!|\delta|\!)^{\Lambda} \sqsubseteq \bot$
5. $[\, (\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}} \to \{e\}\,] \sqsubseteq \partial_1$.

Point (1) and induction imply that $\Gamma; \partial_2 \vdash_\ell P'$. Point (5), the hypothesis $\partial_1 \sqsubseteq \partial_2$ and Proposition 4.2(1) imply that $e \in \mathtt{Priv}_\Gamma(\partial_2, \ell'')$, where $(\!|\ell'|\!)_\ell = \ell''$. Point (4) is equivalent to $\partial \sqsubseteq (\!|\delta|\!)^{\Lambda}$; by Proposition 4.2(3) and induction, this implies that $\Gamma; (\!|\delta|\!)^{\Gamma}_{\ell} \vdash_{\ell''} Q$. By Proposition 4.2(3) and point (3), we have that $(\!|\delta|\!)^{\Gamma}_{\ell} = (\!|\delta|\!)^{\Lambda} \sqsubseteq \prod_{\lambda \in (\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}}} \Delta(\lambda) \sqsubseteq \prod_{\lambda \in (\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}}} \pi_2(\Gamma(\lambda)) = \mathtt{Pol}_\Gamma(\ell'')$; the latter inequality holds because, by the hypothesis $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot$ (that is assumed to build Γ), $\Delta(k) \sqsubseteq e\delta$ for every $k ::^{e\delta} P$ in N. By construction of Γ, this implies that $\Delta(k) \sqsubseteq \pi_2(\Gamma(k))$, since $\pi_2(\Gamma(k)) = \bigsqcap_{k ::^{e\delta} P \text{ in } N} e\delta$. We have all the premises of the typing rule for action **eval**; hence, $\Gamma; \partial_2 \vdash_\ell P$, as desired.

Assume that $P = \mathbf{in}(T)@\ell'.P'$; by the premises of the rules for action prefixing and action **in**, we have that

1. $(\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P' : \partial_1, \bot$
2. $\hat{\sigma} \models^{(\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}}}_{1} T : \hat{T}[(\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}}] \triangleright \hat{W}$
3. $[\, (\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}} \to \{i\}\,] \sqsubseteq \partial_1$.

Point (1) and induction imply that $\Gamma; \partial_2 \vdash_\ell P'$. Point (3), the hypothesis $\partial_1 \sqsubseteq \partial_2$ and Proposition 4.2(1) imply that $i \in \mathtt{Priv}_\Gamma(\partial_2, \ell'')$, where $(\!|\ell'|\!)_\ell = \ell''$. Point (2), Propositions 4.2(2) and 4.3(1) imply that $check_{\ell''}(\Gamma, (\!|T|\!)_\ell)$. Indeed, whenever $\hat{U} = \hat{T}[(\!|\ell'|\!)^{\Lambda}_{\hat{\sigma}}] = \Gamma[(\!|\ell'|\!)_\ell]$, we have that the set $K = \{et \in \hat{U} : match((\!|T|\!)_\ell, et) \text{ is defined}\}$ is such that $\pi_i(K) \subseteq \Gamma(u)$, for every i such that $\pi_i(T) =! u$; this suffices to conclude $check_{\ell''}(\Gamma, (\!|T|\!)_\ell)$. Thus, we can conclude $\Gamma; \partial_2 \vdash_\ell P$, as desired. ∎

Proposition 4.5 ("If" part of Theorem 4.1). *If* $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \bot$ *then* $\Gamma \vdash N$.

Proof: The proof is by induction on the length of the inference for $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot$. We have two possible base cases:

- $N = l :: \langle et \rangle$: in this case, we have that $\{et\} \subseteq \hat{T}(l)$. By Proposition 4.2(2), this implies that $et \in \pi_1(\Gamma(l))$ and, hence, $\Gamma \vdash N$.
- $N = l ::^{e\delta} P$: in this case, we have that
  1. $(\hat{T}, \Delta, \hat{\sigma}) \models^{\{l\}}_{\mathrm{P}} P : \partial, \varrho$
  2. $\partial \setminus_{\{l\}} e\delta \sqsubseteq \bot$

3. $\Delta(l) \setminus_{\{l\}} e\delta \sqsubseteq \bot$
4. $\varrho \sqsubseteq \bot$.

Point (4) implies that $\varrho = \bot$ and, similarly, points (2,3) imply that $\partial \sqsubseteq e\delta$ and $\Delta(l) \sqsubseteq e\delta$. By Lemma 4.4, we have that $\Gamma; e\delta \vdash_l P$. Moreover, by construction of Γ, it holds that $\pi_2(\Gamma(l)) \sqsubseteq e\delta$; this suffices to conclude that $\Gamma \vdash l ::^{e\delta} P$, as desired.

The inductive step is trivial. ■

Conversely, we now prove that typeable processes and nets are policy conformant; this suffices to prove the "only if" part of Theorem 4.1. 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:

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

To define Δ, we have to take, for every locality, the least upper bound of all the policies specified for sandboxes at that locality. To this aim, we first need to remove every occurrence of **self** as target of **eval** actions in N as follows (we only give the cases where the function is not the identity):

$$\begin{array}{rclrcl} (\!|N_1 \,\|\, N_2|\!) & = & (\!|N_1|\!) \,\|\, (\!|N_2|\!) & (\!|l ::^{e\delta} P|\!) & = & l ::^{e\delta} (\!|P|\!)_l \\ (\!|P_1 \mid P_2|\!)_\ell & = & (\!|P_1|\!)_\ell \mid (\!|P_2|\!)_\ell & (\!|{*P}|\!)_\ell & = & *\,(\!|P|\!)_\ell \\ (\!|\alpha.P|\!)_\ell & = & (\!|\alpha|\!)_\ell.(\!|P|\!)_\ell & (\!|\mathbf{eval}(Q:\delta)@\ell'|\!)_\ell & = & \mathbf{eval}((\!|Q|\!)_{(\!|\ell'|\!)_\ell} : \delta)@(\!|\ell'|\!)_\ell \end{array}$$

Then, for every $l \in \mathsf{Loc}$, we have to calculate the least upper bound of the policies argument of **eval** actions whose target is l or a variable that can assume value l:

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

Lemma 4.6. *If $\Gamma; \partial \vdash_\ell P$ and $\Lambda = \mathtt{val}_\Gamma(\ell)$, then*

1. $(\hat{T}, \Delta, \hat{\sigma}) \models^\Lambda_P P : \partial, \bot$
2. *for every* $\mathbf{eval}(Q:\delta)@\ell'$ *in P, it holds that* $(\!|\delta|\!)^\Gamma_l \sqsubseteq \pi_2(\Gamma(l))$*, for every* $l \in \mathtt{val}_\Gamma(\ell')$ *(for the sake of compactness, we shall write the previous claim as* $\mathsf{PredEval}(P, \Gamma)$*).*

Proof: By induction on the inference for $\Gamma; \partial \vdash_\ell P$. The base step is trivial; for the inductive step, we only give the most complex cases.

Assume that $P = \mathbf{eval}(Q:\delta)@\ell'.P'$; by the premises of the rule for action **eval**, we have that

1. $(\!|\ell'|\!)_\ell = \ell''$
2. $e \in \mathtt{Priv}_\Gamma(\partial, \ell'')$
3. $(\!|\delta|\!)^\Gamma_\ell = \partial' \sqsubseteq \mathtt{Pol}_\Gamma(\ell'')$
4. $\Gamma; \partial' \vdash_{\ell''} Q$
5. $\Gamma; \partial \vdash_\ell P'$

Points (4,5) and induction imply that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}} Q : \partial', \bot$ and $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^\Lambda P' : \partial, \bot$. Points (1,2) and Proposition 4.2(1) imply that $[(\!|\ell'|\!)^\Lambda_{\hat{\sigma}} \rightarrow \{e\}\,] \sqsubseteq \partial$. Point (3) and Proposition 4.2(3) imply that $(\!|\delta|\!)^\Lambda = \partial'$ and, hence, $\partial' \backslash_{(\!|\ell|\!)^\Lambda_{\hat{\sigma}}} (\!|\delta|\!)^\Lambda \sqsubseteq \bot$. To obtain $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^\Lambda P : \partial, \bot$ it suffices to prove that, for every $\lambda \in (\!|\ell'|\!)^\Lambda_{\hat{\sigma}}$, it holds that $(\!|\delta|\!)^\Lambda \sqsubseteq \Delta(\lambda)$. By construction of Δ, we have considered $(\!|\delta|\!)^\Gamma_\lambda$ when defining $\Delta(\lambda)$, for every $\lambda \in (\!|\ell'|\!)^\Lambda_{\hat{\sigma}}$; since $\Delta(\lambda)$ has been defined as a least upper bound, because of Proposition 4.2(3) we have that $(\!|\delta|\!)^\Lambda \sqsubseteq \Delta(\lambda)$.

We are left with proving $\mathsf{PredEval}(P, \Gamma)$. By point (3), we have that $(\!|\delta|\!)^\Gamma_\ell \sqsubseteq \mathtt{Pol}_\Gamma(\ell'') = \bigsqcap_{\lambda \in \mathtt{val}_\Gamma(\ell'')} \pi_2(\Gamma(\lambda)) \sqsubseteq \pi_2(\Gamma(\lambda))$, where the last inequality holds for every $\lambda \in \mathtt{val}_\Gamma(\ell'')$ by definition of greatest lower bound. This fact, together with $\mathsf{PredEval}(Q, \Gamma)$ and $\mathsf{PredEval}(P', \Gamma)$ (that hold by points (4,5) and induction), suffices to conclude.

Assume that $P = \mathbf{in}(T)@\ell'.P'$; by the premises of the rule for action **in**, we have that

1. $(\!|\ell'|\!)_\ell = \ell''$
2. $i \in \mathtt{Priv}_\Gamma(\partial, \ell'')$
3. $check_{\ell''}(\Gamma, (\!|T|\!)_\ell)$
4. $\Gamma; \partial \vdash_\ell P'$

Point (4) and induction imply that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^\Lambda P' : \partial, \bot$ and $\mathsf{PredEval}(P', \Gamma)$; hence, $\mathsf{PredEval}(P, \Gamma)$ holds as well. Points (1,2) and Proposition 4.2(1) imply that $[(\!|\ell'|\!)^\Lambda_{\hat{\sigma}} \rightarrow \{i\}\,] \sqsubseteq \partial$. Point (3), Propositions 4.2(2) and 4.3(2) imply that $\hat{\sigma} \models_1^{(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}} T : \hat{T}[(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}] \triangleright \hat{W}$, for some $\hat{W}$. Indeed, whenever $\hat{U} = \hat{T}[(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}] = \Gamma[(\!|\ell'|\!)_\ell]$, we have that the set $K = \{et \in \hat{U} : match((\!|T|\!)_\ell, et) \text{ is defined}\}$ is such that $\sigma \sqsubseteq \hat{\sigma}$, for every σ such that $\sigma = match((\!|T|\!)_\ell, et)$ for some $et \in K$; this suffices to conclude $\hat{\sigma} \models_1^{(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}} T : \hat{T}[(\!|\ell'|\!)^\Lambda_{\hat{\sigma}}] \triangleright K$. ∎

Proposition 4.7 ("Only if" part of Theorem 4.1). *If* $\Gamma \vdash N$ *then* $(\hat{T}, \Delta, \hat{\sigma}) \models_N N : \bot$.

Proof: The proof is by induction on the length of the inference for $\Gamma \vdash N$. We have two possible base cases:

- $N = l :: \langle et \rangle$: in this case, we have that $et \in \pi_1(\Gamma(l))$. By Proposition 4.2(2), this implies that $\{et\} \subseteq \hat{T}(l)$ and, hence, $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{N}} N : \bot$.
- $N = l ::^{e\delta} P$: in this case, we have that

1. $\Gamma; e\delta \vdash_l P$
2. $\pi_2(\Gamma(l)) \sqsubseteq e\delta$.

Point (1) and Lemma 4.6(1) imply that $(\hat{T}, \Delta, \hat{\sigma}) \models_{\mathrm{P}}^{\{l\}} P : e\delta, \bot$ and, clearly, $e\delta \backslash_{\{l\}} e\delta \sqsubseteq \bot$. Point (2) and Lemma 4.6(2) imply that $\Delta(l) \backslash_{\{l\}} e\delta \sqsubseteq \bot$. Indeed,

$$\begin{aligned}
\Delta(l) = & \bigsqcup_{l' ::^{e\delta} R \ in \ N} \bigsqcup_{\mathbf{eval}(Q:\delta)@\ell \ in \ (\!|R|\!)_{l'} \ : \ l \in \mathtt{val}_\Gamma(\ell)} (\!|\delta|\!)_l^\Gamma \\
\sqsubseteq & \bigsqcup_{l' ::^{e\delta} R \ in \ N} \bigsqcup_{\mathbf{eval}(Q:\delta)@\ell \ in \ (\!|R|\!)_{l'} \ : \ l \in \mathtt{val}_\Gamma(\ell)} \pi_2(\Gamma(l)) \\
\sqsubseteq & \ e\delta
\end{aligned}$$

The inductive step is trivial. ∎

4.4. 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 4.1 (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 Γ makes the running example typeable:

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

for every $K \in \{U, B, C\}$ and $x \in \{title, data\}$.

4.5. 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 the evaluated policy resulting from replacing **self** with the actual locality in δ. 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. Dynamic Creation of Localities

Having presented the Flow Logic and the type system for our extension of KLAIM, we are now ready to consider the **newloc** action, which allows processes to dynamically extend systems structure by creating new localities. The main challenge when dealing with this construct is how the policies have to be modified to take into account the new locality. We investigate here an approach where **newloc** takes three arguments, namely a variable u that is bound to the new locality name, a set C of capabilities that describes the capabilities that the creating locality should get with respect to the new locality, and a policy δ' that describes the policy associated to the newly created locality.

In the rest of this section, we first present the semantics of the **newloc** action and then show how to handle it with the Flow Logic and with the type system, respectively. We conclude with a short discussion on how to handle more sophisticated versions of the **newloc** action.

5.1. Semantics

To account for the creation of new localities, and to ensure their uniqueness, we change the form of the semantic reduction rules from $L \triangleright N \longrightarrow N'$ to $L \triangleright N \longrightarrow L' \triangleright N'$.

We add the following reduction rule for **newloc**:

$$\frac{l' \notin L \qquad \overline{l'} = \overline{u} \qquad (\!|\delta'|\!)_{l'} = e\delta' \qquad \mathsf{RM}[e\delta(l) \ni n]}{L \triangleright l ::^{e\delta} \mathbf{newloc}(u : C, \delta').P \longrightarrow L \cup \{l'\} \triangleright l ::^{e\delta[l' \mapsto C]} P[u \mapsto l'] \,\|\, l' ::^{e\delta'} \mathbf{nil}}$$

The $\mathbf{newloc}(u : C, \delta')$ action creates a new locality with a fresh name in the system. The name is bound to the variable u declared in the **newloc** action, thereby allowing the creating process to access and communicate with the newly created locality as well as sending it to other processes in the system.

The decorated localities $\overline{l'}$ and $\overline{u}$ in the premise of the rule above are used to denote *canonical* representatives. We do assume that each locality and locality variable, say ℓ, belongs to a family containing infinitely many localities with canonical representative $\overline{\ell}$. Hence, condition $\overline{l'} = \overline{u}$ ensures that variables can be instantiated at runtime only by localities 'of the same kind'. This condition is essential for the Flow Logic to compute sound estimates of all possible future behaviours of the system without knowing in advance the exact localities created during execution. The condition does not impose any severe requirement on the semantics and indeed later we argue that it can be safely ignored (and, hence, removed) when introducing the type system.

5.2. Reconsidering the Running Example

We reconsider the example presented in Subsection 2.2 and show how to make use of the **newloc** action. We extend the example with a new process P_{BL} running at l_B that offers a special service to loyal customers, by which they can get books out of the library directly. Let us assume that l_U is a loyal customer, running process P_{UL}, and add to the system the sub-net

$$l_B ::^{e\delta_{BL}} P_{BL} \,\|\, l_U ::^{e\delta_{UL}} P_{UL}$$

The processes are:

$$\begin{array}{rcl} P_{UL} & = & \mathbf{in}(!clubloc)@\mathbf{self}.\mathbf{out}(clubloc, \mathbf{self}, \text{J.R.R. Tolkien})@l_{BS}. < await\ response > \\ P_{BL} & = & \mathbf{newloc}(club : \{r, o\}, [\,]).\mathbf{out}(club)@l_U.\mathbf{out}(l_U)@club. \\ & & \mathbf{in}(club, !member, !request)@\mathbf{self}.\mathbf{read}(member)@club. < handle\ request > \end{array}$$

The access policies of nodes are:

$$\begin{array}{rcl} e\delta_{UL} & = & [l_U \mapsto \{i\},\ l_B \mapsto \{o\}] \\ e\delta_{BL} & = & [l_B \mapsto \{i, n\}, l_U \mapsto \{o\}] \end{array}$$

The process P_{BL} creates a new locality stored in variable *club* that it shares with loyal customers, and uses it to check whether a request comes from any such customer (again, in this simple example we have omitted other, possibly disloyal, costumers). The only requirement is that l_{LC} and l_{BS} have the capability to output tuples at each others locality. Of course, for executing activity $< handle\ request >$, l_{BL} might need more capabilities (according to the kind of the activity), but that is beyond the scope of this example.

5.3. Flow Logic

We now show how to extend the Flow Logic from Section 3 to handle **newloc**. This extension mostly amounts to exploiting canonical localities, that is replacing Loc with $\overline{\mathsf{Loc}}$. As a consequence of this, the analysis domains must be updated to reflect the use of canonical localities:

- $\hat{T} \in \overline{\mathsf{Loc}} \to \mathcal{P}(\overline{\mathsf{Loc}}^*)$
- $\hat{\sigma} \in \mathsf{LocVar} \to \mathcal{P}(\overline{\mathsf{Loc}})$
- $\partial \in \mathsf{AbstractPolicy} = \overline{\mathsf{Loc}} \to \mathcal{P}(\mathsf{Capabilities})$
- $\Delta \in \overline{\mathsf{Loc}} \to \mathsf{AbstractPolicy}$
- $\varrho \in \overline{\mathsf{Loc}} \to \overline{\mathsf{Loc}} \to \mathcal{P}(\mathsf{Capabilities})$

- $\Lambda \in \mathcal{P}(\overline{\mathsf{Loc}})$

The rule for **newloc** to be added to the Flow Logic specification differs in format from the rules used previously in that we, for the sake of simplicity, decide to analyze the continuation process together with the **newloc** itself.

$$\frac{\overline{u} \in \hat{\sigma}(u) \qquad [\Lambda \to \{n\}] \sqsubseteq \partial \qquad \Delta(\overline{u}) \setminus_{\{\overline{u}\}} (\!|\delta|\!)^{\{\overline{u}\}} \sqsubseteq \varrho \qquad (\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} P : \partial', \varrho \qquad \partial'[\overline{u} \to \partial'(\overline{u}) \setminus C] \sqsubseteq \partial}{(\hat{T}, \Delta, \hat{\sigma}) \models^{\Lambda}_{\mathrm{P}} \mathbf{newloc}(u : C, \delta).P : \partial, \varrho}$$

In $\hat{\sigma}$ we record that the locality variable u is associated with the canonical name $\overline{u}$ and then we use $\overline{u}$ in the environment entries. The premise $[\Lambda \to \{n\}] \sqsubseteq \partial$ ensures that all localities where this action might be executed record the n capability. The premise $\Delta(\overline{u}) \setminus_{\{\overline{u}\}} (\!|\delta|\!)^{\{\overline{u}\}} \sqsubseteq \varrho$ ensures that we correctly record potential violations that might arise from processes remotely executed on the newly created locality. The fact that we analyze the continuation process in the rule that deals with the **newloc** action itself makes it easy to modify the analysis result arising from the continuation process: the actions of C are permitted for any process running at the creating locality, so they should be removed as potential errors for processes running at that locality – this is expressed by the premise $\partial'[\overline{u} \to \partial'(\overline{u}) \setminus C] \sqsubseteq \partial$.

To ensure soundness of the analysis result we require evaluated policies to be *canonical consistent* as defined below.

Definition 5.1 (Canonical Consistence). *An evaluated policy $e\delta$ is* canonical consistent *iff $\overline{l} = \overline{l'} \Rightarrow e\delta(l) = e\delta(l')$. A net is* canonical consistent *if all evaluated policies occurring in it are canonical consistent.*

When a canonical consistent net evolves, canonical consistence is preserved if

- in the original net, no **newloc**$(u : C, \delta)$ has a value of $\overline{u}$ that equals the canonical name $\overline{l}$ of any locality l in the net or some $\overline{u'}$ for any *other* **newloc**$(u' : C', \delta')$ occurring in the net, and
- in the original net, no distinct localities have the same canonical name, i.e. $l \neq l' \Rightarrow \overline{l} = \overline{l'}$,

In the analysis of processes, this allows us to avoid computing the most restrictive or most permissive policy for analyzing processes, as the evaluated policies coincide for all newly created localities.

The results proved for the analysis presented in Section 3.8 hold also in the enhanced framework; the **newloc** has no dramatic impact on the proofs of such results, that are left to the interested reader.

5.4. Type System

The type system can be easily adapted to deal with the form of **newloc** discussed so far. First of all, like in the Flow Logic approach we have to work with canonical localities, that are locality sorts and are associated by the type system to the localities and variables occurring in the analized net. Thus, we assign types to canonical localities and to variables: types for canonical localities are pairs $\langle \mathcal{T}; \partial \rangle$, where $\mathcal{T} \subset_{\text{fin}} \overline{\mathsf{Loc}}^*$ and $\partial \in \mathsf{AbstractPolicy} = \overline{\mathsf{Loc}} \to \mathcal{P}(\mathsf{Capabilities})$; types for input variables are sets of canonical localities $\mathcal{T} \subset_{\text{fin}} \overline{\mathsf{Loc}}$. Of course, we now have that $\mathtt{val}_\Gamma(l) = \{\bar{l}\}$ and we still have that $\mathtt{val}_\Gamma(u) = \Gamma(u)$; consequently,

$$\Gamma\langle \ell \rangle = \bigcap_{\bar{l} \in \mathtt{val}_\Gamma(\ell)} \pi_1(\Gamma(\bar{l})) \qquad \mathtt{Priv}_\Gamma(\partial, \ell) = \bigcap_{\bar{l} \in \mathtt{val}_\Gamma(\ell)} \partial(\bar{l})$$

$$\Gamma[\ell] = \bigcup_{\bar{l} \in \mathtt{val}_\Gamma(\ell)} \pi_1(\Gamma(\bar{l})) \qquad \mathtt{Pol}_\Gamma(\ell) = \bigsqcap_{\bar{l} \in \mathtt{val}_\Gamma(\ell)} \pi_2(\Gamma(\bar{l}))$$

The typing rule for the **newloc** can be defined as follows:

$$\frac{\overline{u} \in \Gamma(u) \qquad n \in \mathtt{Priv}_\Gamma(\partial, \ell) \qquad (\!|\delta|\!)^\Gamma_u \sqsubseteq \mathtt{Pol}_\Gamma(u) \qquad \Gamma; \partial[\overline{u} \mapsto C] \vdash_\ell P}{\Gamma; \partial \vdash_\ell \mathbf{newloc}(u : C, \delta).P}$$

The main result of our paper, the accordance between the two analyses (Theorem 4.1), holds also in this new setting. To establish this result, it suffices to add one more inductive case to the proofs of Lemmas 4.4 and 4.6.

It is worth noticing that, while the premise "$\overline{l'} = \overline{u}$" in the operational rule for the **newloc** presented in Section 5.1 is needed for the Flow Logic analysis, it could be removed when considering the type system. We have kept it to take advantage of the results already proved for the Flow Logic. If we get rid of it, we could still use the type system as presented so far, but we would need to explicitly prove the subject reduction theorem. Such a result would be formulated as follows:

If N_1 is typeable and $L_1 \triangleright N_1 \longrightarrow_{\mathsf{off}} L_2 \triangleright N_2$, then N_2 is typeable.

Notice that, if $L_2 = L_1$, we can routinely prove that $\Gamma \vdash N_1$ implies $\Gamma \vdash N_2$. Otherwise, it must be $L_2 = L_1 \cup \{l'\}$, for some fresh l' that has been created by some l as the result of a **newloc**$(u : C, \delta)$; in this case, we prove that $\Gamma \vdash N_1$ implies $\Gamma' \vdash N_2$, where Γ' extends Γ in the following way:

- $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{\bar{l'}\}$;
- $\Gamma'(\bar{\ell}) = \begin{cases} \Gamma(\bar{\ell}) & \text{if } \bar{\ell} \neq l' \\ \langle \Gamma\langle u \rangle; \mathtt{Pol}_\Gamma(u) \rangle & \text{otherwise} \end{cases}$

Then, we can freely let $\overline{u}$ be the canonical locality associated to l' and this would make the premise "$\overline{l'} = \overline{u}$" useless in the semantics for **newloc**.

5.5. Final Remarks

The version of **newloc** described in this section only supports modifications of the policies at the creating and at the created localities. There are no provisions for granting any right to the other localities in the net relatively to the newly created one. A possibility to generalize our approach would take the form of broadcasting capabilities throughout the system (possibly in a 'controlled' way) when a new locality is created.

To add broadcasting to the Flow Logic specification, one would need to add a component very similar to the Δ component in Section 3, which records policies for remotely executed processes as shown in Figure 5. In the case of **eval**, it made sense to compute the most permissive policy of all possible targets of the remote execution; in the case of **newloc**, it would be more sensible to compute the most restrictive policy. This may be achieved by letting the Flow Logic calculate the set of relevant policies and then take the intersection. The technical details in pursuing this approach are somewhat complex in the general case, but become tractable when we take advantage of the restriction to canonical consistent policies.

As expected from what we already observed, the type system can be easily tailored for accommodating the handling of the 'broadcasting' version of the **newloc**. Indeed, the typing rules remain the same. We just need to change part of the proof of the subject reduction theorem of Section 5.4 in the case of a **newloc**: when passing from Γ to Γ', we add the entry for the newly created node (as shown before) and update the existing information with the capabilities C over the canonical locality associated to the new node.

6. Conclusions and Future and Related Work

We have considered an extension 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 existing type systems for access control with some dynamic checks and, by exploiting concepts from the Flow Logic, 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 the two static approaches are sound with respect to the dynamic one based on a reference monitor and provide the same analysis results.

Future work. We see this work just as an initial step towards understanding the relationships between static and dynamic approaches to access control and towards studying the

relative merit of type systems and Flow Logic specifications (expanding on [25]). In future work, we intend to study the relationships between the global approach of static type systems (and Flow Logic) and the more local one of type systems with dynamic checks. Moreover, we find it challenging to understand the relative expressive power of reference monitors and static analysis approaches also in light of the considerations of [36], 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. Finally, it would also be interesting to make the security model assumed in this paper more powerful, by following some directions already taken for KLAIM via type systems with dynamic type checks [15, 16]. In particular, it would be challenging to allow dynamically evolving policies (by means of capability passing, loss, expiration and/or removal) as in [15], or policies for migrating agents that depend on the "source" site, as in [16].

Related work. In literature, there are many papers proposing static analysis techniques for process calculi, mostly based on or inspired by type systems. Due to space limitations, we mention here only some of the most recent ones for process calculi with distribution and process mobility. For type systems, we would like to mention [6, 8, 9, 19, 21]. In all these papers, a type is assigned to the communication medium (either a located channel [19] or an ambient [6, 8, 9, 21]) for regulating the data exchanges in every computation. This is similar to our type-based approach. Indeed, also in our case every communication medium (i.e., tuple space) is assigned a type that describes the kind of data that can be placed there. The main difference between these approaches and ours is that our types are in principle less prescriptive, in the sense that different kinds of data can appear in the same tuple space; on the contrary, channels and ambients usually host just one kind of data (actually, this condition is sometimes relaxed by exploiting subtyping). Of course, the greater freedom of our types is compensated by the check performed by function *match*.

The use of canonical localities somehow resembles the use of abstract names in [22]. Both notions are used to group together names (of localities in our case and of ambients in theirs) which can be assigned the same type. This turns out to be very useful in calculi where names can be dynamically created. Also in the type system of [22], sets of abstract names are used to overapproximate the behaviour of a system and the sort of data that can appear within an ambient. The approach based on group types adopted in another paper on the Ambient calculus [7] is similar.

Flow Logics have been developed over the last decade as an approach for combining insights from Data Flow Analysis, Control Flow Analysis, Abstract Interpretation and Type and Effect Systems [28]. The approach has been used for analysing a wide variety of programming languages exhibiting a variety of functional, imperative, or object-oriented features; we refer to [30] for an overview.

Only recently Flow Logic has been extended to deal with calculi of computation with concurrent, distributed and mobile features. It has been used to analyse security properties in concurrent calculi, as for example the π-calculus [3] and the LySa calculus [2, 5], viz. a variant of π-calculus with cryptographic primitives. Also several variants of the Ambient calculus have been analysed with focus on security properties, see e.g. [26, 24, 13, 4, 31]; in particular, the latter presents various access control policies. We would also like to mention [23], that studies security problems in the context of wireless networks as represented by a calculus with broadcast.

Finally, there is a large body of work sharing with ours the aim of relating different static approaches to program analysis and type systems for the same language. Most notably, previous work has focused on control-flow analysis and type systems for higher-order functional languages (see e.g. [33, 18, 32, 34]) or on data-flow analysis and type systems for imperative languages [20]. To the best of our knowledge, our work is the first considering a language with concurrency, distribution and process mobility.

Acknowledgements. We would like to thank Doug Lea and Gianluigi Zavattaro for inviting us to submit our work at this special issue of the Science of Computer Programming Journal devoted to *Coordination'08* and the two anonymous reviewers for their fruitful comments.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13:347–390, 2005.

[3] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis for the π-calculus with applications to security. *Information and Computation*, 168:68–92, 2001.

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

[5] M. Buchholz, H. R. Nielson, and F. Nielson. A calculus for control flow analysis of security protocols. *International Journal of Information Security*, 2:145–167, 2004.

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

[7] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2000.

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

[9] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Information and Computation*, 201(1):1–54, 2005.

[10] 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.

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

[12] R. De Nicola, D. Gorla, R. R. Hansen, F. Nielson, H. R. Nielson, C. W. Probst, and R. Pugliese. From flow logic to static type systems for coordination languages. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008*, volume 5052 of *Lecture Notes in Computer Science*, pages 100–116. Springer, 2008.

[13] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In J. He and M. Sato, editors, *Advances in Computing Science - ASIAN 2000, 6th Asian Computing Science Conference*, volume 1961 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2000.

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

[15] D. Gorla and R. Pugliese. Dynamic management of capabilities in a network aware coordination language. To appear in *Journal of Logic and Algebraic Programming*. An extended abstract appear with title "Resource Access and Mobility Control with Dynamic Privileges Acquisition" in *ICALP'03*.

[16] D. Gorla and R. Pugliese. Enforcing security policies via types. In D. Hutter, G. Müller, W. Stephan, and M. Ullmann, editors, *Security in Pervasive Computing, First International Conference*, volume 2802 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2004.

[17] R. R. Hansen, C. W. Probst, and F. Nielson. Sandboxing in myKlaim. In *Proceedings of the The First International Conference on Availability, Reliability and Security, ARES 2006, The International Dependability Conference - Bridging Theory and Practice*, pages 174–181. IEEE Computer Society, 2006.

[18] N. Heintze. Control-flow analysis and type systems. In A. Mycroft, editor, *Proceedings of the Second International Symposium on Static Analysis (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 1995.

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

[20] P. Laud, T. Uustalu, and V. Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theor. Comput. Sci.*, 364(3):292–310, 2006.

[21] F. Levi and D. Sangiorgi. Mobile safe ambients. *ACM Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.

[22] C. Lhoussaine and V. Sassone. A dependently typed ambient calculus. In D. A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2004.

[23] S. Nanz and C. Hankin. A framework for security analysis of mobile wireless networks. *Theor. Comput. Sci.*, 367(1-2):203–227, 2006.

[24] F. Nielson, R. R. Hansen, and H. R. Nielson. Abstract interpretation of Mobile Ambients. *Science of Computer Programming*, 47:145–175, 2003.

[25] F. Nielson and H. R. 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.

[26] F. Nielson, H. R. Nielson, and R. R. Hansen. Validating firewalls using flow logics. *Theoretical Computer Science*, 283(2):381–418, 2002.

[27] F. Nielson, H. R. Nielson, H. Sun, M. Buchholtz, R. R. Hansen, H. Pilegaard, and H. Seidl. The Succinct Solver Suite. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, number 2988 in Lecture Notes in Computer Science, pages 251–265. Springer, 2004.

[28] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, Berlin, Germany, second edition, 2005.

[29] F. Nielson, H. Seidl, and H. R. Nielson. A succinct solver for ALFP. *Nord. J. Comput.*, 9(4):335–372, 2002.

[30] H. R. Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. In T. Mogensen, D. Schmidt, and I. H. Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer, 2002.

[31] H. R. Nielson, F. Nielson, and M. Buchholtz. Security for Mobility. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design II*, volume 2946 of *Lecture Notes in Computer Science*, pages 207–266. Springer, 2004.

[32] J. Palsberg. Equality-based flow analysis versus recursive types. *ACM Trans. Program. Lang. Syst.*, 20(6):1251–1264, 1998.

[33] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, 17(4):576–599, 1995.

[34] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Program.*, 11(3):263–317, 2001.

[35] 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.

[36] F. 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.

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