Static Validation of Licence Conformance Policies

René Rydhof Hansen
Aalborg University
rrh@cs.aau.dk

Flemming Nielson Hanne Riis Nielson Christian W. Probst
Technical University of Denmark *
{nielson,riis,probst}@imm.dtu.dk

Abstract

Policy conformance is a security property gaining importance due to commercial interest like Digital Rights Management. It is well known that static analysis can be used to validate a number of more classical security policies, such as discretionary and mandatory access control policies, as well as communication protocols using symmetric and asymmetric cryptography. In this work we show how to develop a Flow Logic for validating the conformance of client software with respect to a licence conformance policy. Our approach is sufficiently flexible that it extends to fully open systems that can admit new services on the fly.

1 Introduction

According to a study commissioned by the Motion Picture Association of America (MPAA), U.S. film studios lost an estimated $6.1 billion in 2005 due to piracy. Internet piracy alone was estimated to account for 38% of the losses [3]. While such studies are notoriously difficult to perform and the numbers fraught with uncertainty and controversy, they do show that managing intellectual property in the Internet age is a serious problem with huge financial consequences. Furthermore, with the growing interest in service-oriented architectures and grid computing, where the logical boundaries between individual computers are even more blurred than today, the problems relating to Digital Rights Management (DRM), and policy enforcement in general, are likely to grow in both numbers and complexity.

In this paper we show how *static analysis* can be used as a step towards solving the *licence conformance* problem, which can be seen as one of the most important issues in policy enforcement. Checking a process for licence conformance means to verify whether it can be trusted to obey restrictions imposed by a policy, *e.g.*, that a certain information may only be used or copied a certain number of times. We call such information that is subject to a certain policy *managed information*. We demonstrate how static analysis can be used to solve the policy conformance problem in general, by developing an analysis to check code for being conforming with a *specific* policy. The policy that we exemplarily enforce prohibits both unauthorised copying of managed information, and propagating managed information to actors that are *not* licence conforming.

Our analysis is able to track *linearity properties* of managed information. Linearity properties essentially guarantee that for managed information no more than a specified number of copies exist at any point in time, and that client software will never attempt to copy or use any managed information in an unauthorised manner, *i.e.*, that the client software is licence, and thereby policy, conforming.

We develop the static analysis for a variant of the Klaim calculus [4] that provides a natural, high-level model for distributed systems, *e.g.*, service-oriented architectures and grid computing, and serves as a good foundation for studying different notions of security in such systems [5, 2]. The analysis is developed as a *staged* Flow Logic specification, leveraging a previously developed control-flow analysis for Klaim [2], systematically extending it with linearity effects.

Many approaches to policy enforcement and static analysis require availability of the whole program or system at "compile time", thereby severely restricting the usefulness of these techniques in real-world systems. These are usually *open* systems, where parts of the system are only provided at "run-time", *e.g.*, through plug-ins or dynamic downloads. This is particularly true for service-oriented architectures and grid computing, where a program may only be assembled at run-time by composing a number of services. We believe that the "whole program" requirement is anathema to the idea of open and dynamic systems, and show how our approach can be modularised to work in an open system and thus dispense with the "whole program" requirement.

The rest of this paper is structured as follows. In the next section we give a brief overview of the process calculus used as a formal model. In Section 3 we define what it means for information to be managed securely during execution of a system (at run-time). This is lifted to the

*This work was partially supported through the IST-2005-16004 Integrated Project SENSORIA: Software Engineering for Service-Oriented Overlay Computers.

compile-time level in Section 4, which presents our main contribution, the Flow Logic specification for a static analysis to enforce a licence conformance policy *before* execution. In Section 5 we extend our approach to deal with open, dynamically changing systems. Finally, Section 6 concludes the paper.

2 Syntax and Semantics

The Klaim family of process calculi [1, 4], which forms the basis for this work, is motivated by and designed around the tuple space paradigm, in which a system consists of distributed nodes that interact and communicate through shared tuple spaces by asynchronously sending and receiving tuples. Remote evaluation of processes is used to model mobility.

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 specified in Figure 1. A net consists of processes or tuples located at a location l, or a composition of two nets. Processes can be composed from three base elements—action prefixing, parallel composition, or the invocation of a process declaration. The actual building blocks of processes are the actions: $\mathbf{out}$ ($\mathbf{in}$ and $\mathbf{read}$) actions allow to output (input) tuples to (from) another location's tuple space; while the $\mathbf{in}$ action deletes the read tuple, the $\mathbf{read}$ action is non-destructive. The $\mathbf{eval}$ action models mobility by allowing to send a process to another location, where it will be evaluated, and the $\mathbf{newloc}$ action allows to create new locations, *e.g.*, to model private tuple spaces. To these standard Klaim actions we add the $\mathbf{use}$ action to model the use of data, *e.g.*, documents.

Having introduced the syntax of the calculus, we look at how to define tuples used for communication. As shown in Figure 2, we distinguish between *tuples* and *evaluated tuples*. An *evaluated tuple* is a sequence of values, and can be stored in tuple spaces. In contrast, *tuples* are allowed to contain variables and are used to compose data to be communicated. Note that for succinctness we use localities to represent both actual localities and data values.

When inputting tuples from tuple spaces, processes need to select, which tuple to read or input. This filtering is performed by means of *templates* T that are similar to tuples, but can also contain input variables and *self*-references (explained below), denoted $!u$ and $\mathbf{self}$, respectively. We shall assume throughout that templates are well-formed in the sense that they do not contain both u and $!u$ for the same locality variable u. The matching proceeds by comparing a template component-wise with an evaluated tuple.

Localities			
ℓ	$::=$	l	locality
	$\mid$	$\mathbf{self}$	self
	$\mid$	u	locality variable
Nets			
N	$::=$	$l :: P$	process
	$\mid$	$l :: \langle et \rangle$	located tuple
	$\mid$	$N_1 \parallel N_2$	net composition
Processes			
P	$::=$	$\mathbf{nil}$	nil process
	$\mid$	$a.P$	action prefixing
	$\mid$	$P_1 \mid P_2$	parallel composition
	$\mid$	A	process invocation
Actions			
a	$::=$	$\mathbf{out}(t)@\ell$	output
	$\mid$	$\mathbf{in}(T)@\ell$	input
	$\mid$	$\mathbf{read}(T)@\ell$	read
	$\mid$	$\mathbf{eval}(P)@\ell$	migration
	$\mid$	$\mathbf{newloc}(u)$	creation
	$\mid$	$\mathbf{use}(u)$	use data
Process Declaration			
D	$::=$	$A \stackrel{\triangle}{=} P$	process declaration

Figure 1. Syntax for the Klaim calculus.

t	$::=$	$\ell \mid \ell, t$	tuples
et	$::=$	$l \mid l, et$	evaluated tuple
T	$::=$	$F \mid F, T$	templates
F	$::=$	$\ell \mid !u$	template fields

Figure 2. Syntax for tuples and templates.

For space reasons we do neither present the semantics of the pattern matching nor the Klaim calculus, but refer to [1, 4]. Here we only discuss the extension of Klaim with a notion of $\mathbf{self}$ that enables self-reference in remote-evaluated code. To realise this, we introduce the function $(\!|\cdot|\!)_l : (\mathsf{Loc} \cup \{\mathbf{self}\}) \times \mathsf{Loc} \to \mathsf{Loc}$ that is used to evaluate locations in the semantics relative to the location l where a process is executed: If $(\!|\ell|\!)_l$ is invoked at location l it evaluates to l if $\ell = \mathbf{self}$, and to ℓ otherwise. We trivially extend the $(\!|\cdot|\!)_l$ function from working on single locations to working on templates with all location variables substituted by concrete locations, by defining it to act as the identity on input variables. Similarly, we extend it to sequences in a component-wise manner.

3 Dynamic Policy Conformance

In order to use the Klaim calculus for modelling systems with policy conformance issues, we must first define what

it means to obey to or to violate a policy. In this work, we exemplarily look at policies that restrict the use and replication of information. In the following we give a definition of what it means for information to be copied and used in an unauthorised manner, and formalise the notion of information that is *managed securely*, *i.e.*, in such a way that no unauthorised copying or use takes place. This constitutes that the data is used conforming to our policy.

Since we use locations for modelling both data, *e.g.*, documents, and actual locations, *e.g.*, hosts, it is convenient to identify a subset of *managed locations*, $\mathcal{L} \subseteq \overline{\mathsf{Loc}}$, representing everything that must be tracked for licence conformance purposes. Additionally we may want to model locations that are free as to how they handle managed locations. Therefore we identify a subset of *privileged locations*, $\mathcal{P} \subseteq \mathsf{Loc}$.

A minor technical complication is that we use Loc to denote the set of location constants as used in the semantics, and $\overline{\mathsf{Loc}}$ to denote a set of *canonical* location constants. Each location constant $l \in \mathsf{Loc}$ can be mapped to a canonical location constant $\bar{l} \in \overline{\mathsf{Loc}}$, and similarly each location variable $u \in \mathsf{LocVar}$ can be mapped to a canonical location constant $\overline{u} \in \overline{\mathsf{Loc}}$ (essentially saying that $\overline{\mathsf{Loc}} = \overline{\mathsf{LocVar}}$). This machinery is needed because the location constants can be created dynamically in the semantics.

Ensuring that a managed location is not used or copied improperly is essentially a matter of *counting* how many copies of a given managed location there are in the system of interest. To this end we inductively define functions for counting the number of actual occurrences of managed locations in a net, process, template, or field: $\mathcal{C}^{\mathcal{L}}_{\mathrm{N}}$, $\mathcal{C}^{\mathcal{L},l}_{\mathrm{P}}$, $\mathcal{C}^{\mathcal{L},l}_{\mathrm{T}}$, and $\mathcal{C}^{\mathcal{L},l}_{\mathrm{F}}$, respectively. The counting function, which we omit for space reasons, is mostly a straightforward definition that counts the actual occurrences of a given locality in the semantics. The only noteworthy case is for the **use** action, where the counting function takes the *point-wise maximum* of the counts of the argument to the **use** action and the continuation process, respectively. In this way the counting function will count multiple **use**-actions as only a *single* use of the actual document used, which is the intended semantics.

Before formalising the notion of securely managed information we first need to be able to refer to the location at which a particular action is taken. Based on the semantics of Klaim, every semantic step in a *net*, *i.e.*, a step of the form $L \vdash N \longrightarrow L' \vdash N'$ must be derived from an *action* step, *i.e.*, a step of the form $L \vdash l :: a.P \longrightarrow L' \vdash l :: P$. We say that the former semantic step (the net step) is *rooted at location* l. We can now define the notion of a *securely managed step*. This notion captures the property that copying a managed resource must not take place at a managed location, *i.e.*, every semantic step taken at a managed location must preserve (or at least not increase) the total number of managed locations. Consequently, we call a semantic step $L \vdash N \longrightarrow L' \vdash N'$ rooted at location l a *securely managed step* if $\bar{l} \in \mathcal{L}$ implies that $\mathcal{C}^{\mathcal{L}}_{\mathrm{N}}(N') \sqsubseteq \mathcal{C}^{\mathcal{L}}_{\mathrm{N}}(N)$.

Intuitively a step is managed securely if it does not give rise to an overall increase in the number of managed entities, *i.e.*, an action performed by a process at a managed location must not create additional managed entities. The notion of a securely managed step is easily extended to cover *securely managed sequences* of semantic steps. Finally, a system model that can perform only securely managed sequences is said to be *policy conforming*. In the next section, we show how to guarantee this property by static analysis.

4 Static Policy Conformance

We shall say that a location constant is *managed in a linear manner* if the number of occurrences of it never increases during the execution of a net. We therefore demand that the number of occurrences of elements from the set $\mathcal{L}$ of managed locations never increases *due to actions performed at* $\mathcal{L}$, *i.e.*, in a securely managed step in the sense of the previous section.

4.1 Static Licence Conformance for Nets

In order to formalise these ideas we shall use analysis judgements that take the following form for nets N:

$$(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\mathrm{N}} N : \varrho \,\&\, \gamma, \psi$$

The intention is that when true, the components $\hat{T}$, $\hat{\sigma}$, ϱ, $\mathcal{L}$, γ, and ψ correctly capture not only the behaviour of the net N, but also the behaviour of all the nets it may evolve into. The information captured by the approximation is given by the following components:

- $\hat{T} \in \overline{\mathsf{Loc}} \to \mathcal{P}(\overline{\mathsf{Loc}}^*)$ is an *abstract tuple space*; it keeps a record of all tuples (of location constants) that may at some point reside in the tuple space of a given location constant.
- $\hat{\sigma} \in \mathsf{LocVar} \to \mathcal{P}(\overline{\mathsf{Loc}})$ is an *abstract environment*; it keeps a record of all location constants that a given location variable might at some point be bound to.
- $\varrho \in \overline{\mathsf{Loc}} \times \overline{\mathsf{Loc}} \to \mathcal{P}(\mathsf{Actions})$ is the *action effect*; it records all the actions that may have been performed during the evolution of the net; the first component of the argument is the location where the action was initiated and the second component of the argument is the location where the action had effect and the result of the mapping is a subset of $\mathsf{Actions} = \{o, i, r, e, n\}$ recording which actions actually were performed. The domain of action effects is indeed a complete lattice equipped with the ordering $\sqsubseteq$ defined in a point-wise manner from the subset ordering on $\mathsf{Actions}$, and we write $\sqcup$ for its least upper bound operation.

$$
\begin{array}{l}
(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{N}} l :: P : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad (\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\{\bar{l}\}} P : \varrho \,\&\, \gamma, \psi \\
(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{N}} l :: \langle et \rangle : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad (\!|et|\!)_{\hat{\sigma}}^{\{\bar{l}\}} \subseteq \hat{T}(\bar{l}) \\
(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{N}} N_1 \parallel N_2 : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad (\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{N}} N_1 : \varrho_1 \,\&\, \gamma_1, \psi_1 \\
\qquad (\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{N}} N_2 : \varrho_2 \,\&\, \gamma_2, \psi_2 \\
\qquad \varrho_1 \sqcup \varrho_2 \sqsubseteq \varrho \\
\qquad \gamma_1 \oplus \gamma_2 \sqsubseteq \gamma \\
\qquad \psi_1 \cup \psi_2 \subseteq \psi
\end{array}
$$

Figure 3. Analysis of static licence conformance for nets.

$$
\begin{array}{l}
(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\Lambda} \mathbf{nil} : \varrho \,\&\, \gamma, \psi \\
(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\Lambda} A : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad A \triangleq P \\
\qquad (\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\Lambda} P : \varrho \,\&\, \gamma, \psi \\
(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\Lambda} P_1 \,|\, P_2 : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad (\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\Lambda} P_1 : \varrho_1 \,\&\, \gamma_1, \psi_1 \\
\qquad (\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\Lambda} P_2 : \varrho_2 \,\&\, \gamma_2, \psi_2 \\
\qquad \varrho_1 \sqcup \varrho_2 \sqsubseteq \varrho \\
\qquad \gamma_1 \oplus \gamma_2 \sqsubseteq \gamma \\
\qquad \psi_1 \cup \psi_2 \subseteq \psi
\end{array}
$$

Figure 4. Analysis of licence conformance for processes (part 1).

- $\mathcal{L} \subseteq \overline{\mathsf{Loc}}$ is the set of *canonical location constants* that we want to ensure are managed in a linear manner (managed locations).
- $\gamma \in \mathsf{LocVar} \to \mathbb{N} \cup \{\infty\}$ is a *counting environment*; for each location variable it will record an upper bound on the number of times one of the localities bound to it has been used. Here we use ∞ to record that it was not possible for the analysis to determine an upper bound on the number of uses of the location variable.
- $\psi \subseteq \mathcal{L}$ is an *error component*; it records a subset of locations from $\mathcal{L}$ where the linearity requirement might have been violated.

The aim of the analysis is to ensure an empty error component; this motivates defining that the net N satisfies *static licence conformance* with respect to $\mathcal{L}$, if there exist $\hat{T}, \hat{\sigma}, \varrho, \gamma$ such that $(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{N}} N : \varrho \,\&\, \mathbf{0}, \emptyset$.

The definition of the judgement $(\hat{T},\hat{\sigma}) \models_{\mathrm{N}} N : \varrho$ is given by the inference system of Figure 3. As is usual in Flow Logic we perform a component-wise definition. In the rule for composition of nets we combine the effect contributions from the two subnets using the least upper bound operator and insist that it is included in the action effect of the combined construct. The counting environments are equipped with a partial ordering $\sqsubseteq$ defined by $\gamma_1 \sqsubseteq \gamma_2$ if and only if $\forall u : \gamma_1(u) \leq \gamma_2(u)$ where $\leq$ is the comparison operator on $\mathbb{N}$ extended so that $n \leq \infty$ for all $n \in \mathbb{N} \cup \{\infty\}$. We shall write $\gamma_1 \oplus \gamma_2$ for the counting environment with $(\gamma_1 \oplus \gamma_2)(u) = \gamma_1(u) + \gamma_2(u)$ for all u; here $+$ is the addition operator on $\mathbb{N}$ extended to $\mathbb{N} \cup \{\infty\}$ in the obvious manner. In the case of located tuples we make use of an auxiliary function $(\!|\cdot|\!)_{\hat{\sigma}}^{\Lambda} : \mathsf{Loc} \cup \{\mathbf{self}\} \cup \mathsf{LocVar} \to \mathcal{P}(\overline{\mathsf{Loc}})$ for mapping locations ℓ into the set of canonical locations that they denote. This is straightforward for location constants. In the case of $\mathbf{self}$ we need to supply the set, Λ, of canonical location constants that $\mathbf{self}$ can stand for, and finally in the case of location variables we make use of the abstract environment, $\hat{\sigma}$. This operation is extended to tuples t by taking the cartesian product of all components; in the case of evaluated tuples et we have $(\!|et|\!)_{\hat{\sigma}}^{\Lambda} = \{et\}$.

4.2 Static Licence Conformance for Processes and Actions

The rules make use of the auxiliary judgement for processes $(\hat{T},\hat{\sigma},\mathcal{L}) \models_{\mathrm{P}}^{\Lambda} P : \varrho \,\&\, \gamma, \psi$ defined in Figure 4 and Figure 5; the new component is:

- $\Lambda \in \mathcal{P}(\overline{\mathsf{Loc}})$ is the set of canonical location constants where the process of interest might be encountered during execution.

Because of the treatment of recursive declarations (Figure 4) the definition does *not* constitute a traditional inductive definition. To ensure that we obtain the desired judgement we therefore insist on a *co-inductive* interpretation. As follows from Tarski's fixpoint theorem the inductive and coinductive interpretation coincide if the inference system is defined compositionally in the syntax of nets and processes; in our case the rule for unfolding of recursive constants is the only offending rule.

For action prefixes (Figure 5) we make use of a number of auxiliary notations that will be explained in the following. The context information captured in Λ is only modified in the rule for $\mathbf{eval}(P')@\ell$ where it is updated to be the canonical location constants that ℓ might denote, *i.e.*, $(\!|\ell|\!)_{\hat{\sigma}}^{\Lambda}$.

To more easily express that the appropriate record of actions is captured by the effect component ϱ we shall be using the notation

$$
\begin{array}{lcl}
[X \mapsto C] & : & \overline{\mathsf{Loc}} \times \overline{\mathsf{Loc}} \to \mathcal{P}(\mathsf{Actions}) \\
[X \to C](\bar{l}_1,\bar{l}_2) & = & \begin{cases} C & \text{if } (\bar{l}_1,\bar{l}_2) \in X \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
(\hat{T},\hat{\sigma},\mathcal{L}) \models^{\Lambda}_{\mathrm{P}} \mathbf{out}(t)@\ell.P : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad (\!|t|\!)^{\Lambda}_{\hat{\sigma}} \subseteq \hat{T}\langle (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \rangle \\
\qquad (\hat{T},\hat{\sigma},\mathcal{L}) \models^{\Lambda}_{\mathrm{P}} P : \varrho' \,\&\, \gamma', \psi \\
\qquad [\Lambda \times (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \mapsto \{o\}\,] \sqcup \varrho' \sqsubseteq \varrho \\
\qquad \gamma' \oplus \Gamma^{\oplus}_{t} \sqsubseteq \gamma \\
(\hat{T},\hat{\sigma},\mathcal{L}) \models^{\Lambda}_{\mathrm{P}} \mathbf{in}(T)@\ell.P : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad \hat{\sigma} \models_1^{(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}} T : \hat{T}[(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}] \triangleright \hat{W} \\
\qquad (\hat{T},\hat{\sigma},\mathcal{L}) \models^{\Lambda}_{\mathrm{P}} P : \varrho' \,\&\, \gamma', \psi' \\
\qquad [\Lambda \times (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}} \mapsto \{i\}\,] \sqcup \varrho' \sqsubseteq \varrho \\
\qquad (\gamma' \oplus \Gamma^{\oplus}_{T})\backslash_{\mathsf{bn}(T)} \sqsubseteq \gamma \\
\qquad \psi' \subseteq \psi \\
\qquad \Lambda \cap \mathcal{L} \neq \emptyset \Rightarrow \\
\qquad\quad \{u \in \mathsf{bn}(T) \mid \\
\qquad\qquad \hat{\sigma}(u) \cap \mathcal{L} \neq \emptyset \wedge \gamma'(u) > 1\} \subseteq \psi \\
(\hat{T},\hat{\sigma},\mathcal{L}) \models^{\Lambda}_{\mathrm{P}} \mathbf{use}(u).P : \varrho \,\&\, \gamma, \psi \\
\quad \text{iff} \quad (\hat{T},\hat{\sigma},\mathcal{L}) \models^{\Lambda}_{\mathrm{P}} P : \varrho' \,\&\, \gamma', \psi' \\
\qquad \varrho' \sqsubseteq \varrho \\
\qquad \psi' \subseteq \psi \\
\qquad \max(\Gamma^{\oplus}_{u}, \gamma') \sqsubseteq \gamma
\end{array}
$$

Figure 5. Analysis of licence conformance for processes (part 2): Action prefixes for selected actions.

where $\bar{l}_1$ denotes the canonical location constant where the action might be initiated, $\bar{l}_2$ denotes the canonical location constant where the action might have effect, and C usually is a singleton set. In the case of $\mathbf{out}$, $\mathbf{in}$, $\mathbf{read}$ and $\mathbf{eval}$ we choose $X = \Lambda \times (\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}$, so we record the location of the process in the first component and the location effected in the second component. In the case of $\mathbf{newloc}(u)$ we use the canonical location variable $\overline{u}$ to indicate the point where the action has effect.

Since most of the rules need to take effect for any element in some set Y of canonical location 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 combinators so as to reduce the explicit use of quantifiers. The notations are formally defined by:

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

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

$$
\begin{array}{rcl}
\Psi[Y] \subseteq X & \iff & \forall y \in Y : \Psi(y) \subseteq X \\
X \subseteq \Psi\langle Y \rangle & \iff & \forall y \in Y : X \subseteq \Psi(y)
\end{array}
$$

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

In the rules for prefixing we need to count the occurrences of the various location variables. We write $\Gamma^{\oplus}_{T}$ for the counting environment that maps the location variable $u \in \mathsf{LocVar} \cup \{\mathbf{self}\}$ to the number of times it occurs free in T:

$$\Gamma^{\oplus}_{T}(u) = k \quad \text{if } u \text{ occurs } k \text{ times in } T$$

We use this operation to add up the number of occurrences of the location variables in the rule for $\mathbf{out}(t)@\ell$: clearly if u occurs in t then this represents a use of the location constant bound to u.

In the cases of $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$ we shall also make sure that the contributions of $\Gamma^{\oplus}_{T}$ are added to those of the continuation. Surely this only accounts for uses of location variables bound in the context; as a result of the matching the location parameters of $\mathsf{bn}(T) = \{u \mid !u \text{ occurs in } T\}$ will get bound: These location variables are of no interest outside their scope so we shall use the notation $\gamma\backslash_{U}$ to stand for the mapping that is as γ, except that all location variables of U have count 0. Hence the rules use $(\gamma' \oplus \Gamma^{\oplus}_{T})\backslash_{\mathsf{bn}(T)} \sqsubseteq \gamma$ to express that these location variables are of no interest outside their scope. For $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$ it is of interest whether the action is performed at a managed location. If $\Lambda \cap \mathcal{L} \neq \emptyset$, then it is of interest whether any of the location variables in the template $(\mathsf{bn}(T))$ can be bound to a managed location ($\hat{\sigma}(u) \cap \mathcal{L} \neq \emptyset$), and how often they are used. In the case of $\mathbf{read}(T)@\ell$ no use is allowed, so the rule contains the premise:

$$\Lambda \cap \mathcal{L} \neq \emptyset \Rightarrow \{u \in \mathsf{bn}(T) \mid \hat{\sigma}(u) \cap \mathcal{L} \neq \emptyset \wedge \gamma'(u) > 0\} \subseteq \psi$$

For $\mathbf{in}(T)@\ell$ it will be allowed, provided that it only occurs once and hence we get the condition

$$\Lambda \cap \mathcal{L} \neq \emptyset \Rightarrow \{u \in \mathsf{bn}(T) \mid \hat{\sigma}(u) \cap \mathcal{L} \neq \emptyset \wedge \gamma'(u) > 1\} \subseteq \psi$$

ensuring that if one of the locations of $\mathcal{L}$ indeed is used for the input then the location parameters cannot be rebound in the continuation. For $\mathbf{use}(u)$ we mimic the operation of the counting function defined in Section 3 and take the pointwise maximum of the argument to $\mathbf{use}$ and the counting environment for the continuation.

It may be worth pointing out, that the analysis only checks for violation of the security policy in the rules for $\mathbf{in}$ and $\mathbf{read}$, and *not* in the rule for $\mathbf{out}$. Even though the security policy demands that a managed entity cannot both be output and retained for subsequent use, it is not necessary to make any checks in the rule for $\mathbf{out}$, because the necessary caution has already been taken care of when binding any offending variable in $\mathbf{out}$ to some $\mathbf{in}$ or $\mathbf{read}$.

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

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

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

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

Figure 6. Analysis of pattern matching: $\hat{\sigma} \models^{\Lambda}_{i} T : \hat{U} \triangleright \hat{W}$**.**

4.3 Pattern Matching

In the rules for $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$ we make use of an auxiliary judgement for performing the pattern matching of tuples against a given pattern. It is given by $\hat{\sigma} \models^{\Lambda}_{i} T : \hat{U} \triangleright \hat{W}$, and expresses that matching should start at position i in the template T. Furthermore, $\hat{U}$ contains the set of tuples that we are matching against, and $\hat{W}$ will then contain the tuples from $\hat{U}$ that successfully match T from position i and onwards. At the same time $\hat{\sigma}$ will record the appropriate bindings that need to be performed. The judgement is defined in Figure 6. Here $\pi_i(et)$ denotes the i'th component of the tuple et, and $\pi_i(\hat{V})$ is the component-wise extension of the operation to sets of tuples.

The judgement is used in the rules for $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$ in Figure 5 to ensure that the matching may succeed. The set of tuples of interest are those of the tuple space of ℓ, that is, $\hat{T}[(\!|\ell|\!)^{\Lambda}_{\hat{\sigma}}]$. Actually, the analysis could be made more precise by only requiring that the analysis result holds for the continuation of the process in the case where $\hat{W} \neq \emptyset$, since otherwise the matching is guaranteed not to be successful.

4.4 Properties of Static Licence Conformance

The theoretical foundation of our approach based on process algebras and Flow Logic [7], allows us to formalise consistency of the analysis as a subject-reduction theorem. Overall correctness of the analysis is formalised as an adequacy result that details the semantic consequences of the static property. Finally, existence of best analysis estimates is formalised as a Moore-family result.

Theorem 1 (Subject Reduction)**.** *If* $L_1 \vdash N_1 \longrightarrow L_2 \vdash N_2$ *and* $(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\text{N}} N_1 : \varrho \ \&\ \mathbf{0}, \emptyset$*, then* $(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\text{N}} N_2 : \varrho \ \&\ \mathbf{0}, \emptyset$*.*

Proof. Note that it makes sense to use $\mathbf{0}$ throughout because any variables becoming free by executing a binding in $\mathbf{in}$ or $\mathbf{read}$ are immediately substituted away.

The proof is by induction on the inference using a few auxiliary results, listed below.

The analysis result is invariant under the structural congruence; that is, if $N_1 \equiv N_2$ then $(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\text{N}} N_1 : \varrho \ \&\ \gamma, \psi$ if and only if $(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\text{N}} N_2 : \varrho \ \&\ \gamma, \psi$.

The analysis of matching is correct; that is, if $\mathrm{match}((\!|T|\!)_l, et) = \sigma, \bar{l} \in \Lambda$, $et \in \hat{U}$, and $\hat{\sigma} \models^{\Lambda}_{1} T : \hat{U} \triangleright \hat{W}$, then $et \in \hat{W}$ and $\forall u \in \mathsf{dom}(\sigma) : \overline{\sigma(u)} \in \hat{\sigma}(u)$. □

Theorem 2 (Adequacy)**.** *If* $L_1 \vdash N_1 \longrightarrow L_2 \vdash N_2$ *and* $(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\text{N}} N_1 : \varrho \ \&\ \mathbf{0}, \emptyset$*, then the step is securely managed.*

Proof. The proof is by inspection of Figure 4 and Figure 5. □

Theorem 3 (Moore Family)**.** *For all nets* N*, the set*

$$\mathcal{Y} = \{(\hat{T}, \hat{\sigma}, \varrho, \mathcal{L}, \gamma, \psi) \mid (\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\text{N}} N : \varrho \ \&\ \gamma, \psi\}$$

is a Moore Family*; i.e.,* $\forall Y \subseteq \mathcal{Y} : \sqcap Y \in \mathcal{Y}$*.*

Proof. Due to the co-inductive definition of $(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\text{N}} N : \varrho \ \&\ \gamma, \psi$ (see Figure 4) the formal proof proceeds by co-induction; we refer to the proof of [6, Theorem 3.13] for a detailed explanation. □

4.5 Example: On-line Movie Distribution

We now illustrate how the analysis works by briefly discussing a small example. The example is an excerpt of a larger example concerned with modelling an on-line movie distribution business. The excerpt, shown in Figure 7, models a situation where the movie distributor (represented by

```
1   iPod:: out("f2", iPod)@dist.
2          in("f2", !film_iPod)@self.
3          use(film_iPod).
4          out("f2", film_iPod)@filesrv.
5          use(film_iPod).
6   dist:: D ≙ in(!nam_req, !user)@self.
7          read(nam_req, !film_req)@shelf.
8          out(nam_req, film_req)@user.
9          D
10  shelf:: ⟨"f1", film_1⟩|⟨"f2", film_2⟩|···|⟨"fn", film_n⟩
11 filesrv:: nil
```

Figure 7. Specification for the example system. Variables are set in italic, while location constants are set in regular font. For readability, trailing nil processes and the $\|$ constructors are omitted.

l	$\hat{T}$(l)
iPod	$\{\langle$"f2", film_2$\rangle\}$
dist	$\{\langle$"f2", iPod$\rangle\}$
shelf	$\{\langle$"f1", film_1$\rangle$, $\langle$"f2", film_2$\rangle\}$
filesrv	$\{\langle$film_2$\rangle\}$

u	$\hat{\sigma}$(u)
nam_{req}	{ "f2" }
$user$	{ iPod}
$film_{req}$	{ film_2}
$film_{\texttt{iPod}}$	{ film_2 }

effects from iPod
(iPod, dist) $\mapsto \{o\}$
(iPod, iPod) $\mapsto \{i\}$
(iPod, filesrv) $\mapsto \{o\}$

effects from dist
(dist, dist) $\mapsto \{i\}$
(dist, shelf) $\mapsto \{r\}$
(dist, iPod) $\mapsto \{o\}$

Figure 8. Analysis results for abstract tuple spaces, environments, and the effect.

line/action		γ_i	ψ_i
iPod			
1	**out**	iPod $\mapsto 1$	$film_{\texttt{iPod}}$
2	**in**	**0**	$film_{\texttt{iPod}}$
3	**use**	$film_{\texttt{iPod}} \mapsto 2$	$\emptyset$
4	**out**	$film_{\texttt{iPod}} \mapsto 2$	$\emptyset$
5	**use**	$film_{\texttt{iPod}} \mapsto 1$	$\emptyset$
-	**nil**	**0**	$\emptyset$
dist			
6	**in**	**0**	$\emptyset$
7	**read**	$nam_{req} \mapsto 1$	$\emptyset$
8	**out**	$nam_{req} \mapsto 1$ $film_{req} \mapsto 1$	$\emptyset$
9	D	**0**	$\emptyset$

Figure 9. Analysis results for the counting environment and the error component. The line numbers refer to the code in Figure 7.

the location dist), wants to make sure that a given customer's video iPod[1] (represented by location iPod) is configured to be licence conforming, i.e., configured to disallow unauthorised copying, before they allow the customer to download and watch movies on that device. In addition to the already mentioned locations, the example comprises further two locations: one representing the movie distributor's film database (location dist), and one representing a fileserver on the customer's intranet (location filesrv).

The full result of analysing the example is shown in Figure 8 and Figure 9. We shall not go into detail with the full result here. Instead we focus on the subset of the result that has direct bearing on the example.

In order to check for licence conformance it is sufficient to check the *error component* of the analysis, denoted ψ in Figure 9. Of particular interest is the error component for location iPod: It indicates that there is a *potential* for a licence violation in the iPod's configuration. A closer examination of the result reveals that the film requested by the customer may be used up to two times (lines 3 and 4) which is a violation of the licence. Therefore it *cannot* be established that the customer's device is licence conforming, and consequently no movies may be downloaded onto that particular device until it is properly reconfigured (and re-validated).

5 Open Systems

As mentioned in the introduction, the increased adoption and widespread use of network-based computing, such as grid computing and service-oriented architectures, poses a major challenge for current approaches to policy enforcement and static analysis. The main challenge lies in the fact that parts of the system may be provided through plug-ins or dynamic downloads and so are only available during execution of the system, *i.e.*, at "run-time". In the following we extend our approach to also cover such *open* systems.

Being based on the Klaim calculus, our calculus already has primitives for creating new locations and for roaming (by means of **eval**) the net. To fully model an open environment we add a new primitive, **accept**, that accepts a new process from the *environment* and makes it part of the Trusted Computing Base. To make this safe we must ensure that only processes that satisfy the imposed security policy are admitted into the system. The semantics of the **accept** primitive is given by

$$\frac{(\hat{T}, \hat{\sigma}, \mathcal{L}) \models_{\mathrm{P}}^{\{l\}} Q : \varrho \,\&\, \mathbf{0}, \emptyset}{L \vdash l :: \mathbf{accept}.P \longrightarrow L \vdash l :: P \,\|\, l :: Q}$$

The above semantic rule formalises that a new process Q can only be accepted at a given location l, if it is indeed stat-

[1] This could be any device for watching movies, e.g., a dedicated home theatre or a personal computer.

ically correct (in the sense of the adequacy result) with respect to the "global" analysis information $\hat{T}, \hat{\sigma}, \mathcal{L}, \varrho$, guaranteeing the conformance of the Trusted Computing Base to the security policy. In effect, we ensure that the operational semantics works as a reference monitor that verifies that newly admitted processes satisfies the security policy. Since the reference monitor implements all necessary checks, the actual analysis of the **accept** action is then rather trivial:

$$(\hat{T}, \hat{\sigma}, \mathcal{L}) \models^{\Lambda}_{\mathrm{P}} \mathbf{accept}.P : \varrho \,\&\, \gamma, \psi$$
$$\text{iff} \quad (\hat{T}, \hat{\sigma}, \mathcal{L}) \models^{\Lambda}_{\mathrm{P}} P : \varrho \,\&\, \gamma, \psi$$

We state without proof that the formal correctness results, as expressed in Theorems 1, 2, and 3 for the analysis of closed systems, carry over. This shows the power of our approach, first used in [2], for enforcing the security of open systems by relying on the reference monitor to make the appropriate checks when the Trusted Computing Base is enlarged.

6 Conclusion

We have shown how to use static analysis to guarantee the enforcement of a licence conformance policy. This is achieved by using a static analysis to validate that client code conforms to the security policy, using the correctness theorems to ensure that validated code can be considered part of the Trusted Computing Base.

Technically, the development consists of developing a Flow Logic for tracking the control flow as well as access operations executed by the code. Semantic correctness is established by means of a subject-reduction result and an adequacy result; finally, the existence of best solutions is guaranteed by a Moore family result. The linearity component of the analysis ensures that selected entities are indeed managed in a linear manner. The analysis, as well as the accompanying counting function only count certain direct uses of the entities subject to a licence-conformance policy. In the terminology of *information flow*, we are dealing with direct flows but do not consider indirect flows.

References

[1] L. Bettini, V. Bono, R. D. Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In C. Priami, editor, *Global Computing*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003.

[2] R. R. Hansen, C. W. Probst, and F. Nielson. Sandboxing in myKlaim. In *The First International Conference on Availability, Reliability and Security, ARES'06*, Vienna, Austria, Apr. 2006. IEEE Computer Society.

[3] LEK Consulting. 2005 piracy data summary. Available from `http://www.mpaa.org/2006_05_03leksumm.pdf`, last visited January, 2007.

[4] R. D. Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.

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

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

[7] H. R. Nielson and F. Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer, 2002.