Sandboxing in myKlaim

René Rydhof Hansen, Christian W. Probst, and Flemming Nielson*
Informatics and Mathematical Modelling
The Technical University of Denmark
2800 Kongens Lyngby, Denmark
email {rrh,probst,nielson}@imm.dtu.dk

Abstract

The μKlaim calculus is a process algebra designed to study the programming of distributed systems consisting of a number of locations each having their own tuple space and collection of mobile processes. Previous work has explored how to incorporate a notion of capabilities to be enforced dynamically by means of a reference monitor. Our first contribution is to describe a sandboxing semantics for the remote evaluation of mobile code; we then develop a succinct flow logic for statically guaranteeing the properties enforced by the reference monitor and hence for dispensing with the overhead of a dynamic reference monitor. Our second contribution is an extension of the calculus to interact with an environment; here processes enter the system from the environment and we develop an entry-condition that is sufficient for ensuring that the resulting system continues to guarantee the properties that would otherwise need to be dynamically enforced by the reference monitor. We call the resulting calculus myKlaim.

1 Introduction

Security is a key consideration when programming distributed systems and it is important to be able to make the security considerations at a relevant level of abstraction. Finding an appropriate level of abstraction has motivated the development of a number of process algebras including variations of the π-calculus [6] and mobile ambients [2]. Security annotations and considerations have been performed for most of these [1, 3].

In this paper we consider μKlaim [4], which belongs to a group of process algebras focusing on the locations (or nodes) in a distributed systems and taking the view that each has a separate tuple space and collection of mobile processes. Communication takes the form of outputting or inputting messages (tuples) from named locations and mobility takes the form of remote evaluation of code.

Extensions to μKlaim have been developed for equipping locations with security policies. As is customary in access control the policies regulate how one principal (traditionally called the subject) may invoke resources belonging to another principal (traditionally called the object): actions are supposed to be blocked by a reference monitor unless the corresponding capability is explicitly permitted by a given capability. In the syntax of myKlaim this takes the form $l ::^{[l' \mapsto \pi]}$ for expressing the policy that allows all processes located at l to perform actions permitted by π to the (resources belonging to the) location l'.

Sandboxing In μKlaim a remote evaluation of the form $l ::^{[l' \mapsto \{\pi\}]}$ $\mathbf{eval}(Q)@l'$ intends to execute the process Q at location l' rather than the current location l. This will be accepted by the reference monitor if the capability e for remote evaluation is granted, i.e. if $e \in \pi$ and will lead to Q being evaluated at l' using the policy currently in place at l'.

Our first contribution is to extend the primitive for remote evaluation with the policy to be enforced on the remote invocation. Hence the capability e now becomes $e[\delta]$ where δ is the policy to be enforced. This corresponds to evaluating Q in a *sandbox* located at l' and is general enough to allow different processes at the same location to have different policies.

At the more technical level we then develop a succinct flow logic for statically guaranteeing the properties enforced by the reference monitor. This means that for statically checked processes one may safely "turn off" the reference monitor and hence dispense with the overhead of a dynamic reference monitor. The analysis is proved correct with respect to the operational semantics using a subject-reduction result.

Open Systems Systems written in myKlaim are essentially closed: there is a fixed set of locations and a fixed

*This work has in part been supported by the EU research project #016004, *Software Engineering for Service-Oriented Overlay Computers*.

Localities			
ℓ	$::=$	l	locality
	$\mid$	u	locality variable
Nets			
N	$::=$	$l ::^{\delta} P$	single node
	$\mid$	$l :: \langle et \rangle$	located tuple
	$\mid$	$N_1 \parallel N_2$	net composition
Processes			
P	$::=$	$\mathbf{nil}$	null 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
	$\mid$	$\mathbf{newloc}(u : \delta)$	creation
	$\mid$	$\mathbf{accept}(\delta)$	admit external proc.

Figure 1. Syntax for processes and actions.

T	$::=$	$F \mid F, T$	templates
F	$::=$	$f \mid !x \mid !u$	template fields
t	$::=$	$f \mid f, t$	tuples
f	$::=$	$e \mid l \mid u$	tuple fields
et	$::=$	$ef \mid ef, et$	evaluated tuple
ef	$::=$	$V \mid l$	evaluated tuple field
e	$::=$	$V \mid x \mid \ldots$	expressions

Figure 2. Tuples and fields .

$$\begin{array}{rcll}
N_1 \parallel N_2 & \equiv & N_2 \parallel N_1 & \\
(N_1 \parallel N_2) \parallel N_3 & \equiv & N_1 \parallel (N_2 \parallel N_3) & \\
l ::^{\delta} P & \equiv & l ::^{\delta} (P \mid \mathbf{nil}) & \\
l ::^{\delta} A & \equiv & l ::^{\delta} P & \text{if } A \stackrel{\triangle}{=} P \\
l ::^{\delta} (P_1 \mid P_2) & \equiv & l ::^{\delta} P_1 \parallel l ::^{\delta} P_2 &
\end{array}$$

Figure 3. Structural congruence.

$$\mathrm{match}(V, V) = \epsilon \qquad \mathrm{match}(!x, V) = [V/x]$$

$$\mathrm{match}(l, l) = \epsilon \qquad \mathrm{match}(!u, l') = [l'/u]$$

$$\frac{\mathrm{match}(F, ef) = \sigma_1 \qquad \mathrm{match}(T, et) = \sigma_2}{\mathrm{match}((F, T), (ef, et)) = \sigma_1 \circ \sigma_2}$$

Figure 4. Tuple matching.

set of policies. While new locations can be constructed and extensions of myKlaim allow to modify the network topology there is no direct account of accepting new code into the system. Such actions might correspond to software updates or downloading applets to the current security domain.

Our second contribution is the extension of myKlaim with an *accept* primitive for allowing new code to enter. Since our aim still is to dispense with the dynamic enforcement of a reference monitor, we modify the operational semantics such that the tests performed upon the code entering are sufficient to guarantee the intended security policy.

2 The myKlaim Calculus

Process calculi of the Klaim family are centered around modelling the *tuple space* paradigm in which a system is comprised by a distributed set of nodes that communicate by outputting and getting tuples from one or more shared tuple spaces. Mobility is modelled through remote evaluation of processes. In this paper we consider a variant of the μKlaim calculus [4] called the *myKlaim* calculus. The myKlaim calculus incorporates a notion of *sandboxing* and primitives for access control. In the basic tuple space paradigm, tuple spaces are shared resources among peers and thus no attempt is made to restrict or control access to these. For the purpose of modelling service-oriented architectures or other network structures that may include mutually distrusting parties it is convenient to extend the calculus with access control primitives. This is achieved by adding a *security policy* to every node with a process and by embedding a *reference monitor* into the operational semantics of the calculus. The reference monitor controls every execution step in order to ensure that the security policy is not violated. In a later section we develop an analysis that can statically guarantee that a program cannot possibly violate the security policy. This result will allow us to remove all the runtime checks performed by the reference monitor, while preserving the security of the system. Security policies are formalised as *capability lists*, i.e., at every locality (node) an associated set of capabilities defines which actions can be performed at that node and which other nodes can be accessed:

$$\begin{array}{rcl}
\pi \subseteq \mathsf{Capability} & = & \{i, r, o, e[\delta], n\} \\
\delta \in \mathsf{Policy} & = & \mathsf{Loc} \to \mathcal{P}(\mathsf{Capability})
\end{array}$$

where i, r, o, e, and n correspond to the actions $\mathbf{in}$, $\mathbf{read}$, $\mathbf{out}$, $\mathbf{eval}$, and $\mathbf{newloc}$, respectively.

Intuitively, a policy δ for a location l determines what processes located at l are allowed to do to *other* locations. Note that the capability for performing an $\mathbf{eval}$-action is associated with a security policy, δ, that specifies the security policy of the *sandbox* that the remotely evaluated process is

$$\frac{\mathrm{match}([\![T]\!], et) = \sigma \qquad \boxed{\mathsf{RM} = \mathsf{on} \Rightarrow i \in \delta(l')}}{l ::^{\delta} \mathbf{in}(T)@l'.P \parallel l' :: \langle et \rangle \rightarrowtail_{\text{RM}} l ::^{\delta} P\sigma \parallel l' :: \mathbf{nil}}$$

$$\frac{[\![t]\!] = et \qquad \boxed{\mathsf{RM} = \mathsf{on} \Rightarrow o \in \delta(l')}}{l ::^{\delta} \mathbf{out}(t)@l'.P \parallel l' ::^{\delta'} P' \rightarrowtail_{\text{RM}} l ::^{\delta} P \parallel l' ::^{\delta'} P' \parallel l' :: \langle et \rangle}$$

$$\frac{L \vdash N_1 \rightarrowtail_{\text{RM}} L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \rightarrowtail_{\text{RM}} L' \vdash N_1' \parallel N_2}$$

$$\frac{\mathrm{match}([\![T]\!], et) = \sigma \qquad \boxed{\mathsf{RM} = \mathsf{on} \Rightarrow r \in \delta(l')}}{l ::^{\delta} \mathbf{read}(T)@l'.P \parallel l' :: \langle et \rangle \rightarrowtail_{\text{RM}} l ::^{\delta} P\sigma \parallel l' :: \langle et \rangle}$$

$$\frac{N \equiv N_1 \qquad L \vdash N_1 \rightarrowtail_{\text{RM}} L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \rightarrowtail_{\text{RM}} L' \vdash N'}$$

$$\frac{\boxed{\mathsf{RM} = \mathsf{on} \Rightarrow e[\delta''] \in \delta(l')}}{l ::^{\delta} \mathbf{eval}(Q)@l'.P \parallel l' ::^{\delta'} P' \rightarrowtail_{\text{RM}} l ::^{\delta} P \parallel l' ::^{\delta'} P' \parallel l' ::^{\delta''} Q}$$

$$\frac{l' \notin L \qquad \lfloor l' \rfloor = \lfloor u \rfloor \qquad \delta' \preceq \delta \qquad \boxed{\mathsf{RM} = \mathsf{on} \Rightarrow n \in \delta(l)}}{L \vdash l ::^{\delta} \mathbf{newloc}(u : \delta').P \rightarrowtail_{\text{RM}} L \cup \{l'\} \vdash l ::^{\delta[l' \mapsto \delta(l)]} P[l'/u] \parallel l' ::^{\delta'[l'/u]} \mathbf{nil}}$$

Figure 5. Reference monitor semantics for myKlaim in closed systems, i.e., *without* the accept-action.

executed in. Later in this section we discuss the sandboxing in more detail.

Like μKlaim, the myKlaim calculus comprises three parts: nets, processes, and actions. Nets give the overall structure in which tuple spaces and processes are located. Processes execute by performing actions. The syntax is shown in Fig. 1 and Fig. 2. The main difference in syntax compared to [4] is that we add an *accept* action for admitting a new process to enter a system.

Processes can be either the **nil**-process doing nothing, or a process that executes an action a, or indeed a parallel composition of subprocesses. Finally, a process can be an "invocation" of a process place-holder variable (used for recursion). The **out**-action outputs a tuple into a tuple space at a specific locality; the **in** and **read** actions input a tuple from a specific tuple space, either removing it or leaving it in place respectively; the **eval**-action remotely evaluates a process at a specified locality (subject to sandboxing); **newloc** creates a new locality; and **accept** injects a process from the environment.

Next, we define and briefly discuss the reference monitor semantics of the myKlaim calculus for *closed* systems, i.e., *without* the accept action. The semantics of the accept primitive will be introduced in Section 5 as part of open systems. Since the semantics is very similar to the semantics for μKlaim, we do not go into details with the "standard" parts of the semantics, but refer to [4].

The reference monitor semantics for myKlaim is obtained directly from the operational semantics of μKlaim by adding a reference monitor. As explained in the introduction, the reference monitor can either be turned on or off; the current state is indicated as a subscript on the semantic arrow, i.e., the RM annotation on $\rightarrowtail_{\text{RM}}$. If the reference monitor is active, then all actions are checked against the relevant security policy before being allowed to execute; in the semantics the checks are displayed in boxes. Fig. 3 and Fig. 4 show the structural congruence of nets and processes and the rules for matching tuples, respectively. These are used in Fig. 5 to define the semantic reduction rules of myKlaim.

Sandboxing is modelled by allowing a single locality to have several different security policies and thereby allowing processes at the same node to execute in separate and different security environments. The only visible change relative to the μKlaim semantics is in the rule for remote evaluation of processes: the **eval**-action.

2.1 An Example

We will use an example similar to subscribing online publications from [4]. In our scenario we have a publisher P that owns two locations—the publisher's reading room l_P and a shelf l_S. P stores all its publications in l_S and users are only allowed to access the data in l_S while they are in the reading room l_P. The typical reader R will evaluate a process at the reading room l_P. This process will read a paper from the shelf and "use" it in some way. As an example, reader R_1 defined by

$$R_1 \triangleq \mathbf{eval}(\mathbf{read}(\text{“paper1”}, !p1)@l_S.\text{use } p1)@l_P$$

sends a process to the reading room, which will get the paper with the tag *paper1* from the shelf and use it. This should be fine with the publisher, as long as the use of the paper does not include the **out** action. A reader R_2, defined by

$$R_2 \triangleq \mathbf{eval}(\mathbf{read}(\text{“paper2”}, !p2)@l_S.\mathbf{out}(p2)@l_{R_2})@l_P$$

should be prohibited by the publisher from entering the reading room, as it reads the data of the paper labeled *paper2* from the shelf and forwards it to its own location l_{R_2}.

$$
\begin{array}{lll}
\hat{\sigma} \models_i \epsilon : \hat{V}_\circ \triangleright \hat{V}_\bullet & \text{iff} & \{\hat{et} \in \hat{V}_\circ \,|\, |\hat{et}| = i-1\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i V, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \mathrm{prj}_i(\hat{et}) = V\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i l, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \mathrm{prj}_i(\hat{et}) = l\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i x, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \mathrm{prj}_i(\hat{et}) \in \hat{\sigma}(x)\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i u, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \mathrm{prj}_i(\hat{et}) \in \hat{\sigma}(u)\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i !x, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \hat{V}_\circ \sqsubseteq \hat{V}_\bullet \wedge \mathrm{prj}_i(\hat{W}_\bullet) \sqsubseteq \hat{\sigma}(x) \\
\hat{\sigma} \models_i !u, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \hat{V}_\circ \sqsubseteq \hat{V}_\bullet \wedge \mathrm{prj}_i(\hat{W}_\bullet) \sqsubseteq \hat{\sigma}(u)
\end{array}
$$

Figure 6. Abstract tuple matching.

In original μKlaim calculus, the only capability granted by the publisher to other localities in the system is to do an **eval** on the reading room l_P. Thus, the capabilities assigned to the different locations in the system would be

$$
\begin{array}{l}
l_{R_1} ::^{[l_{R_1} \mapsto \{*\}, l_P \mapsto \{e\}]} R_1 \parallel l_{R_2} ::^{[l_{R_2} \mapsto \{*\}, l_P \mapsto \{e\}]} R_2 \parallel \\
l_P ::^{[l_S \mapsto \{r\}]} \mathbf{nil} \parallel \\
l_S :: \langle\text{“paper1”}, \mathit{data}_1\rangle | \langle\text{“paper2”}, \mathit{data}_2\rangle | \cdots
\end{array}
$$

We use $*$ to abbreviate that a process running at a location l is allowed to perform all actions at l. The setting above allows both R_1 as well as R_2 to evaluate a process at the reading room l_P (capability $l_P \mapsto \{e\}$), and once their process is executed at l_P, they can read papers from the shelf l_S due to capability $l_S \mapsto \{r\}$. However, to impose the publishers requirement, that processes running in the reading room l_P should not be able to perform the **out** action, the original μKlaim calculus will, as part of the static inference system, rewrite processes and mark those actions that can not be checked statically. In the example this is necessary, since the reading room might dynamically acquire the capability $l \mapsto \{o\}$, which would allow processes running at l_P to communicate with location l. So the network that eventually will be executed contains processes defined by

$$
\begin{array}{l}
\underline{R_1} \overset{\triangle}{=} \mathbf{eval}(\underline{\mathbf{read}}(\text{“paper1”}, !p1)@l_S.\text{use } p1)@l_P \\
\underline{R_2} \overset{\triangle}{=} \mathbf{eval}(\underline{\mathbf{read}}(\text{“paper2”}, !p2)@l_S.\underline{\mathbf{out}}(p2)@l_{R_2})@l_P
\end{array}
$$

The operational semantics of μKlaim will check for the $\underline{\mathbf{read}}$ and $\underline{\mathbf{out}}$ actions that the location that they are executed at actually has the capability to do so. In the example the $\underline{\mathbf{read}}$ will be allowed, but the $\underline{\mathbf{out}}$ action will cause a stall. In contrast, the myKlaim version of the above system is specified by

$$
\begin{array}{l}
l_{R_1} ::^{[l_{R_1} \mapsto \{*\}, l_P \mapsto \{e[l_S \mapsto \{r\}]\}]} R_1 \parallel \\
l_{R_2} ::^{[l_{R_2} \mapsto \{*\}, l_P \mapsto \{e[l_S \mapsto \{r\}]\}]} R_2 \parallel \\
l_P ::^{[]} \mathbf{nil} \parallel l_S :: \langle\text{“paper1”}, \mathit{data}_1\rangle | \langle\text{“paper2”}, \mathit{data}_2\rangle | \cdots
\end{array}
$$

The operational semantics will ensure that the processes evaluated at l_P by R_1 and R_2 respectively are executed in a sandbox that only has the capabilities specified for the **eval** action. After executing both **eval** actions, the system evolves to

$$
\begin{array}{l}
l_{R_1} ::^{[l_{R_1} \mapsto \{*\}, l_P \mapsto \{e[l_S \mapsto \{r\}]\}]} \mathbf{nil} \parallel \\
l_{R_2} ::^{[l_{R_2} \mapsto \{*\}, l_P \mapsto \{e[l_S \mapsto \{r\}]\}]} \mathbf{nil} \parallel l_P ::^{[]} \mathbf{nil} \parallel \\
l_P ::^{[l_S \mapsto \{r\}]} \mathbf{read}(\text{“paper1”}, !p1)@l_S.\text{use } p1 \parallel \\
l_P ::^{[l_S \mapsto \{r\}]} \mathbf{read}(\text{“paper2”}, !p2)@l_S.\mathbf{out}(p2)@l_{R_2} \parallel \\
l_S :: \langle\text{“paper1”}, \mathit{data}_1\rangle | \langle\text{“paper2”}, \mathit{data}_2\rangle | \cdots
\end{array}
$$

Obviously, this system is going to violate the capability specification, since the process reading “paper2” will perform an **out** action. The next section introduces an analysis that statically analyses systems and finds these properties.

3 The Analysis

In this section we specify a static control flow analysis for the myKlaim calculus described in the previous section. The information is collected in three main components:

- $\hat{T}$ is a “global” component recording for each location the set of tuples that can reside in the tuple space belonging to that location;
- $\hat{\sigma}$ is a “global” component recording for each name the set of names it may be bound to (and in the case of constants this will just be the constant itself);
- $\hat{\partial}$ is a “local” that in the case of processes records all actions performed by the process and in the case of nets only those actions not explicitly allowed by the policies governing the processes in the net.

In a later section we show how the analysis can be used to statically guarantee that no runtime violations of the reference monitor can occur in a given net.

$$
\begin{array}{rclcrcl}
\hat{\sigma}[\![f, t]\!] & = & \hat{\sigma}[\![f]\!] \times \hat{\sigma}[\![t]\!] & \quad & \hat{\sigma}[\![l]\!] & = & \{l\} \\
\hat{\sigma}[\![V]\!] & = & \{V\} & & \hat{\sigma}[\![u]\!] & = & \hat{\sigma}(u) \\
\hat{\sigma}[\![x]\!] & = & \hat{\sigma}(x) & & & &
\end{array}
$$

Figure 7. Extension of $\hat{\sigma}$ to templates.

$$\begin{array}{lll}
(\hat{T},\hat{\sigma}) \models_{\text{N}} l ::^{\delta} P : \partial & \text{iff} & (\hat{T},\hat{\sigma}) \models_{\text{P}}^{\lfloor l \rfloor} P : \partial' \wedge \partial' \setminus \{(a,\lfloor l \rfloor) \,|\, a \in \delta(l)\} \subseteq \partial \\
(\hat{T},\hat{\sigma}) \models_{\text{N}} l :: \langle et \rangle : \partial & \text{iff} & \langle et \rangle \in \hat{T}(\lfloor l \rfloor) \\
(\hat{T},\hat{\sigma}) \models_{\text{N}} (N_1 \parallel N_2) : \partial & \text{iff} & (\hat{T},\hat{\sigma}) \models_{\text{N}} N_1 : \partial' \wedge (\hat{T},\hat{\sigma}) \models_{\text{N}} N_2 : \partial'' \wedge \partial' \cup \partial'' \subseteq \partial \\
\hline
(\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} \mathbf{nil} : \partial & \text{iff} & \mathit{true} \\
(\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} P_1 \mid P_2 : \partial & \text{iff} & (\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} P_1 : \partial' \wedge (\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} P_2 : \partial'' \wedge \partial' \cup \partial'' \subseteq \partial \\
(\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} A : \partial & \text{iff} & (\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} P : \partial \quad \text{if } A \triangleq P \\
(\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} a.P : \partial & \text{iff} & (\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} P : \partial' \wedge (\hat{T},\hat{\sigma}) \models_{\text{A}}^{l} a : \partial'' \wedge \partial' \cup \partial'' \subseteq \partial \\
\hline
(\hat{T},\hat{\sigma}) \models_{\text{A}}^{l} \mathbf{out}(t)@\ell : \partial & \text{iff} & \forall l' \in \hat{\sigma}(\ell) :\ \hat{\sigma}[\![t]\!] \subseteq \hat{T}(l') \wedge \{o\} \times \hat{\sigma}(\ell) \subseteq \partial \\
(\hat{T},\hat{\sigma}) \models_{\text{A}}^{l} \mathbf{in}(T)@\ell : \partial & \text{iff} & \forall l' \in \hat{\sigma}(\ell) :\ \hat{\sigma} \models_1 T : \hat{T}(l') \triangleright \hat{W}_{\bullet} \wedge \{i\} \times \hat{\sigma}(\ell) \subseteq \partial \\
(\hat{T},\hat{\sigma}) \models_{\text{A}}^{l} \mathbf{read}(T)@\ell : \partial & \text{iff} & \forall l' \in \hat{\sigma}(\ell) :\ \hat{\sigma} \models_1 T : \hat{T}(l') \triangleright \hat{W}_{\bullet} \wedge \{r\} \times \hat{\sigma}(\ell) \subseteq \partial \\
(\hat{T},\hat{\sigma}) \models_{\text{A}}^{l} \mathbf{eval}(Q)@\ell : \partial & \text{iff} & \forall l' \in \hat{\sigma}(\ell) : (\hat{T},\hat{\sigma}) \models_{\text{P}}^{l'} Q : \partial' \wedge \forall (a,n) \in \partial' : (\{e[n \mapsto \{a\}]\} \times \hat{\sigma}(\ell)) \subseteq \partial \\
(\hat{T},\hat{\sigma}) \models_{\text{A}}^{l} \mathbf{newloc}(u : \delta) : \partial & \text{iff} & \{n\} \times \{l\} \subseteq \partial \wedge \{\lfloor u \rfloor\} \subseteq \hat{\sigma}(\lfloor u \rfloor)
\end{array}$$

Figure 8. Flow logic specification for control flow analysis of myKlaim.

We define the analysis using the Flow Logic framework, cf. [11], that takes a specification oriented approach to determining whether or not a given analysis estimate $\hat{T}, \hat{\sigma}, \partial$ correctly describes all configurations reachable from a given initial net. More concretely, we will be defining judgements for the main syntactic categories in question for axiomatising (i.e. determining the truth of falsity) whether or not the analysis estimate $\hat{T}, \hat{\sigma}, \partial$ is acceptable in the sense that its correctness can be proved with respect to the operational semantics (see Theorem 1 below). Since the form of the correctness result is a subject reduction result this means that analysis estimates may be “too large”. The next step therefore is to use standard techniques (not covered here but see e.g. [10]) for turning the specification into a form where “the least” acceptable analysis estimate can be computed in polynomial time.

Before we continue with a more detailed explanation of the analysis specification we first discuss a technical issue, namely how to handle the dynamic creation of locations. Since unique new locations can be generated during program execution, the analysis would have to keep track of a potentially infinite number of locations. To overcome this and maintain the tractability of the analysis, we define so-called *canonical names* that essentially divide all concrete location names and location variables into equivalence classes in such a way that all (new) location names generated at the same program point belong to the same equivalence class and thus share the same canonical name. The canonical name (equivalence class) of a location or location variable, ℓ, is written $\lfloor \ell \rfloor$. To reduce notational clutter, we shall make use of a *unique representative* for each equivalence class and thereby dispense with the $\lfloor \cdot \rfloor$ notation whenever possible. In order to avoid possible inconsistencies in the security policy for two locations with the same canonical name we shall only consider policies that are *compatible* with the choice of canonical names; we say that a policy is *compatible* with the canonical names if and only if $\lfloor \ell_1 \rfloor = \lfloor \ell_2 \rfloor \implies \delta(\ell_1) = \delta(\ell_2)$. This definition entails that the policy assigns the exact same set of capabilities to all locations with the same canonical name. Throughout this paper we tacitly assume that policies are compatible with the chosen canonical names.

For the control flow analysis of myKlaim we shall shortly explain the separate judgements for nets $(\hat{T},\hat{\sigma}) \models_{\text{N}} N : \partial$, processes $(\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} P : \partial$ and actions $(\hat{T},\hat{\sigma}) \models_{\text{A}}^{l} a : \partial$. The definition (Fig. 8) makes use of the auxiliary judgement for pattern matching $\hat{\sigma} \models_i T : \hat{V}_{\circ} \triangleright \hat{W}_{\bullet}$ defined in Fig. 6 and the extension of $\hat{\sigma}$ to templates defined in Fig. 7. It is important to note that free variables in the right hand sides of a Flow Logic specification are implicitly existentially quantified. The judgement for nets $(\hat{T},\hat{\sigma}) \models_{\text{N}} N : \partial$ proceeds in a syntax-directed manner and checks the acceptability of $\hat{T}, \hat{\sigma}$ on all components. In the case of ∂ obtained from processes we make sure to remove all actions explicitly permitted by the policy of the relevant location. This takes the form of ensuring that ∂ contains a certain set (rather than being equal to a certain set) in keeping with the general methodology explained above. The judgement for processes $(\hat{T},\hat{\sigma}) \models_{\text{P}}^{l} P : \partial$ also proceeds in a mainly syntax-directed manner, except for the need to unfold recursive processes. This does not invalidate our axiomatisation, as in general we take a co-inductive rather than inductive interpretation of a Flow Logic specification (see [11]). The judgement for actions $(\hat{T}, \hat{\sigma}) \models^{l}_{\text{A}} a : \partial$ records the tuples being output into the relevant tuple space, obtains the tuples being input using the judgement for matching, and records the actions performed in ∂. The judgement for matching $\hat{\sigma} \models_i T : \hat{V}_\circ \triangleright \hat{W}_\bullet$ traverses the template in a forward direction (starting at index i that is supposed not to exceed the length of T) and then in a backward direction (stopping at index i). In the forward direction the tuples in $\hat{V}_\circ$ are tested against the relevant component of the template T and only tuples satisfying the requirements are carried forward. In the backward direction the tuples in $\hat{W}_\bullet$ are those that passed all requirements and the values in the relevant component are used for defining the names (of the form $!x$ or $!u$) to be matched in that component. We can now formulate the semantic soundness of the analysis as a subject reduction result:

Theorem 1 (Subject Reduction)**.** *If* $(\hat{T}, \hat{\sigma}) \models_{N} N : \partial$ *and* $L \vdash N \succ\!\!\longrightarrow_{RM} L' \vdash N'$ *then* $(\hat{T}, \hat{\sigma}) \models_{N} N' : \partial$.

Proof. (Sketch) By induction on the structure of $L \vdash N \succ\!\!\longrightarrow_{\text{RM}} L' \vdash N'$ and using auxiliary results for the other judgements. □

3.1 Analysing the Example

Since our example does not make use of any **newloc**'s to create new locations, we can simply define the equivalence classes for canonical names to contain only the locations itself. This allows us to proceed with the example while conveniently ignoring canonical names. Note that security policies are always compatible with this choice.

We now apply the analysis defined above to the example from Section 2.1. The analysis starts at

$$(\hat{T}, \hat{\sigma}) \models_{\text{N}} l_{R_1} ::^{[cap_1]} R_1 \parallel l_{R_2} ::^{[cap_2]} R_2 : \partial$$

We have left out the **nil** process running at l_P as well as the location l_S, since it only contains tuples that are used to initialize $\hat{T}$. The analysis then breaks up the net into the single components running in parallel and starts analyzing reader l_{R_1}: $(\hat{T}, \hat{\sigma}) \models_{\text{N}} l_{R_1} ::^{[l_{R_1} \mapsto \{*\}, l_P \mapsto \{e[l_S \mapsto \{r\}]\}]} R_1 : \partial$, which is equivalent to analyzing the process R_1

$$(\hat{T}, \hat{\sigma}) \models^{l_{R_1}}_{\text{P}} \mathbf{eval}(\mathbf{read}(\text{“paper1”}, !p1)@l_S.\text{use } p1)@l_P : \partial$$

Since using $p1$ will not perform any actions, we ignore it for the rest of this section. Analyzing the **eval** action is equivalent to

$$(\hat{T}, \hat{\sigma}) \models^{l_{R_1}}_{\text{A}} \mathbf{eval}(\mathbf{read}(\text{“paper1”}, !p1)@l_S)@l_P : \partial_1 \wedge \\ \forall (a, n) \in \partial_1 : \{e[n \mapsto \{a\}]\} \times \{l_P\} \subseteq \partial$$

which requires to check the evaluated **read** action at the only possible target location l_P:

$$(\hat{T}, \hat{\sigma}) \models^{l_P}_{\text{P}} \mathbf{read}(\text{“paper1”}, !p1)@l_S : \partial_2 \wedge \\ \hat{\sigma} \models_1 (\text{“paper1”}, !p1) : \hat{T}(l_S) \triangleright \hat{W}_\bullet \wedge \{(r, l_S)\} \subseteq \partial_1$$

Now we need to match the tuple (“paper1”, $!p1$) against $\hat{T}(l_S)$, which contains the tuples located at l_S, namely $\{\langle\text{“paper1”}, data_1\rangle, \langle\text{“paper2”}, data_2\rangle, \cdots\}$. The abstract matching defined in Fig. 6 first selects all the tuples matching “paper1” from $\hat{T}(l_S)$, which later is bound to $p1$. This results in $data_1$ being stored in $\hat{\sigma}(p1)$. Having finished the evaluation of the **read** action we can now propagate the computed information upwards. This results in $\partial_1 = \{(r, l_S)\}$ as result of analyzing the **read** and $\partial = \{(e[l_S \mapsto \{r\}], l_P)\}$ as result of analyzing the **eval**. This captures that R_1 performs a **read** action at the reading room l_P. On the top level, the analysis uses this information to compute the ∂. Using the initial set of capabilities for R_1, $l_{R_1} \mapsto \{*\}, l_P \mapsto \{e[l_S \mapsto \{r\}]\}$, this results in $\partial = \emptyset$. Since on the net level the ∂ is an error component, this result indicates that R_1 does not perform an action that would violate any capabilities. Analyzing the process R_2 is almost analogous—with the exception of the **out** action. Analyzing the initial **read** action results in $\partial_3 = \{(r, l_S)\}$ and $data_2 \sqsubseteq \hat{\sigma}(p2)$. Analyzing the **out** action is equivalent to $(\hat{T}, \hat{\sigma}) \models^{l_{R_2}}_{\text{P}} \mathbf{out}(p2)@l_{R_2} : \partial_5$ which results in $\hat{\sigma}[\![t]\!] \subseteq \hat{T}(l_{R_2}) \wedge \{(o, l_{R_2})\} \subseteq \partial_5$. This represents the transfer of $data_2$ to the tuple space of l_{R_2} as result of the **out** action. The sets $\{(o, l_{R_2})\}$ and $\{(r, l_S)\}$ are then propagated to the **eval** action, where they result in $\partial_6 = \{(e[l_S \mapsto \{r\}], l_P), (e[l_{R_2} \mapsto \{o\}], l_P)\}$. This result is used in analysing the initial judgement for the network

$$(\hat{T}, \hat{\sigma}) \models^{l_{R_2}}_{\text{P}} R_2 : \partial_6 \wedge \partial_6 \backslash \{(a, l) | a \in \delta(l)\} \subseteq \partial_7$$

Since the initial set of capabilities for R_2 is $l_{R_2} \mapsto \{*\}, l_P \mapsto \{e[l_S \mapsto \{r\}]\}$ this results in $\partial_8 = \{(e[l_{R_2} \mapsto \{o\}], l_P)\}$. In the next section we will use exactly this property of $\partial \neq \emptyset$ to identify nets that perform unsecure operations. This will be used to define which nets are acceptable.

4 Secure Nets

Informally a net, N, is said to be *dynamically secure* if and only if no execution sequence starting from N can lead to a runtime violation of the reference monitor. This can be formalised as follows:

Definition 2 (Dynamic Security)**.** *A net N (using localities L) is* dynamically secure *if and only if for all L, N and L', N' such that $L \vdash N \succ\!\!\longrightarrow^*_{\mathsf{off}} L' \vdash N'$ we have $L \vdash N \succ\!\!\longrightarrow^*_{\mathsf{on}} L' \vdash N'$.*

Thus for a secure net it does not matter whether the reference monitor is activated or not.

Since the control flow analysis specified in the previous section computes an over-approximation of all the possible flows in a myKlaim program, it can be be used to statically check whether a given program is secure or not. This is formalised below as *static security*:

$$\frac{\mathsf{RM} = \mathsf{off} \Longrightarrow ((\hat{T},\hat{\sigma}) \models_{\mathrm{P}}^{\lfloor l \rfloor} Q : \partial \wedge \partial \subseteq \{(a, \lfloor l \rfloor) \mid a \in \delta'(l)\})}{(\hat{T},\hat{\sigma}), L \vdash l ::^{\delta} \mathbf{accept}(\delta').P \xrightarrow{Q}\!\!{}_{\mathrm{RM}}\ L' \vdash l ::^{\delta} P \parallel l ::^{\delta'} Q}$$

$$(\hat{T},\hat{\sigma}) \models_{\mathrm{A}}^{l} \mathbf{accept}(\delta) : \partial \quad \text{iff} \quad true$$

Figure 9. The upper rule specifies the semantics of the accept-action. It interacts with the environment and receives a process that is executed in a sandbox unless the reference monitor is turned off in which case the new process must pass a static test based on the control flow analysis. The lower part is the flow-logic specification for analysing the action.

Definition 3 (Static Security). *A net, N, is* statically secure *if* $(\hat{T},\hat{\sigma}) \models_{N} N : \partial$ *and* $\partial = \emptyset$.

It is then easy to show that static security is sufficient for also guaranteeing dynamic security:

Theorem 4. *A net that is statically secure is also dynamically secure.*

Proof. (sketch) Follows from subject reduction and inspection of the reference monitor rules. □

As a consequence, any myKlaim program that has been shown to be statically secure will never (attempt to) violate the reference monitor, and therefore all the runtime checking done by the reference monitor can be dispensed with. In addition to better performance this also provides a higher degree of assurance, since a statically secure program is known to *never* violate the security policy—in any possible execution possible. In contrast, a reference monitor will only detect an attempted security violation when it is attempted.

5 Open Systems

In the preceding sections we have focused on myKlaim in a *closed system* scenario, i.e., where all processes in the system are known *a priori*. For highly dynamic network structures, such as grid or service-oriented architectures, that are continuously adapting to meet rapidly changing requirements, it is impossible to analyse the entire system as a whole, since the system is constantly changing and evolving. In this section we show how to extend the previous results to also cover *open systems* where processes can be added to the system dynamically through a special **accept**-action. This special action interfaces with the environment and accepts a process from the environment that is subsequently inserted and executed in a sandbox at the accepting node. Incorporating this change of underlying world-view or paradigm into the formal developments of the previous sections is relatively simple and straightforward since those developments were constructed in anticipation of this change.

First we need to specify the semantics of the **accept**-action as shown in Fig. 9. The action accepts a process Q from the environment (represented by the annotation on the semantic arrow: $\xrightarrow{Q}_{\mathrm{RM}}$). If the reference monitor is *enabled*, the new process is simply placed in a sandbox where it is allowed to run with the policy specified with the **accept**-action. If the reference monitor is *disabled*, the new code is analysed and allowed to run if it complies with the given security policy.

Next, the analysis must be extended to cope with the **accept**-action, but this is trivial, since the actual analysis of the process accepted from the environment is handled in the semantics. In Fig. 9 also the Flow Logic specification for actions is extended to handle the **accept**-action. It is now trivial to prove that the extended analysis retains the subject reduction property:

Theorem 5. *Theorem 1 extends to the new setting.*

Finally, the security property guaranteed by the analysis (as formalised by Theorem 4) also carries over to the extended setting:

Theorem 6. *Theorem 4 extends to the new setting.*

With the above results we the semantics can model *open* systems, while ensuring the security of the system either through sandboxing or by analysing the incoming process in a *Just-in-Time* (JIT) manner and verifying that it cannot possibly violate the given security policy. This flexibility is essential for modelling and verifying highly distributed and dynamic systems.

5.1 The Example Revisited

Using the results in this section we can now lift the restriction on the example that all processes, including all potential readers, must be known beforehand. Instead we can simply let the publisher use an **accept**-action that specifies a suitable policy, e.g., $[l_S \mapsto r]$ specifying that any accepted processes can only read papers from the shelf and nothing else:

$$\begin{array}{l} l_P ::^{[]} \mathbf{accept}([l_S \mapsto r]).\mathbf{nil} \parallel \\ l_S :: \langle \mathit{paper}_1, \mathit{data}_1 \rangle | \langle \mathit{paper}_2, \mathit{data}_2 \rangle | \cdots \end{array}$$

Now the readers can be defined without an explicit $\mathbf{eval}$-action:

$$\begin{array}{lcl} R_1' & \triangleq & \mathbf{read}(\text{“paper1”}, !p1)@l_S.\text{use } p1 \\ R_2' & \triangleq & \mathbf{read}(\text{“paper2”}, !p2)@l_S.\mathbf{out}(p2)@l_{R_2} \end{array}$$

Furthermore, neither of these need to be analysed along with the publisher (the reading room and the shelf), instead this will take place if and when they are accepted into the publishers system. If reader R_1' enters the system it evolves into

$$\begin{array}{l} l_P ::^{[]} \mathbf{nil} \parallel l_P ::^{[l_S \mapsto r]} R_1' \parallel \\ l_S :: \langle \mathit{paper}_1, \mathit{data}_1 \rangle | \langle \mathit{paper}_2, \mathit{data}_2 \rangle | \cdots \end{array}$$

For R_2', if the reference monitor is turned *on*, the system behaves similarly to above, i.e., R_2' is executed in a sandbox of its own. If however the reference monitor is turned *off*, the process will first be analysed, revealing that it may perform an $\mathbf{out}$-action, which is in violation of the security policy. Therefore the process will not be allowed in at all. This demonstrates the flexibility inherent in our approach: the reference monitor can be turned on and off depending on current requirements and the actual situation.

6 Related Work

In [4] access control policies are studied for closed systems modelled in the basic μKlaim-calculus, cf. [7], and a type system is developed for guaranteeing that the security policies are not violated. There is no support for open systems or sandboxing. Further studies of access control in variants of the Klaim calculus can be found in [9, 8].

The concept of sandboxing, in which an untrusted process is allowed to execute in a restricted environment, and variations over this concept, goes back at least to systems research in the early 1970's. The Java virtual machine, cf. [5], provides a recent example of sandboxing.

In [12] sandboxing and security in code mobility is studied in the context of the higher-order π-calculus. Here the approach taken is to show that the security provided by interface types can also be achieved at runtime by using a suitable filtering operator. Furthermore, filter based security is shown to reject fewer safe programs than the corresponding statically typed approach. In contrast, our approach to open systems and code mobility allows the security manager of a system to determine whether the reference monitor should be turned on or off on a case by case basis.

7 Conclusion

In this paper we have developed the myKlaim calculus, a variant of the μKlaim calculus, that is useful for modelling sandboxing and open systems. Furthermore we have specified a control flow analysis for the calculus that can be used to verify that no processes violate the given security policy. Both the analysis and the formal results were extended to cover the full calculus for open systems by using the control flow analysis to perform a just-in-time analysis of incoming code for possible violations of security.

References

[1] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. of the ACM Symposium on Principles of programming languages*, pages 222–235, 2001.

[2] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science (TCS)*, 240(1):177–213, 2000.

[3] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Proc. of Asian Computing Science Conference on Advances in Computing Science*, pages 199–214. Springer-Verlag, 2000.

[4] Gorla and Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2003.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition edition, 1999.

[6] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[7] 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.

[8] R. D. Nicola, G. Ferrari, and R. Pugliese. Types as Specifications of Access Policies. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 117–146. Springer Verlag, 1999.

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

[10] F. Nielson, H. R. Nielson, and H. Seidl. A Succinct Solver for ALFP. *Nordic Journal of Computing*, 2002(9):335–372, 2002.

[11] H. R. Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. In *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer Verlag, 2002.

[12] J. L. Vivas and N. Yoshida. Dynamic channel screening in the higher order pi-calculus. MCS technical report, University of Leicester, May 2002.