Predictive Access Control for Distributed Computation☆

Fan Yang*a, Chris Hankinb, Flemming Nielsona, Hanne Riis Nielsona

aDTU Informatics, Technical University of Denmark, Lyngby, Denmark
bDepartment of Computing and The Institute for Security Science and Technology, Imperial College London, London, United Kingdom

Abstract

We show how to use aspect-oriented programming to separate security and trust issues from the logical design of mobile, distributed systems. The main challenge is how to enforce various types of security policies, in particular predictive access control policies – policies based on the future behavior of a program. A novel feature of our approach is that we can define policies concerning secondary use of data.

Keywords: Coordination Languages, Security Policies, Aspect-oriented Programming, Program Analysis, Tuple Spaces.

1. Introduction

Whilst there is broad agreement that security and other non-functional properties should be designed into systems ab initio it is also recognized that, as society becomes more IT-savvy, our expectations about security and privacy evolve. This is usually followed by changes in regulation in the form of standards and legislation. Thus, although we would still argue that security should feature in the initial design of a system, there is merit in separating out security and other non-functional properties so that they can be updated without disturbing the functional aspects of the system.

This paper focuses on designing a language for specifying policies for access control and direct flow of information. The traditional approach to enforcing such security policies is to use a reference monitor [1] that dynamically tracks the execution of the program; it makes appropriate checks on each basic operation being performed, either blocking the operation or allowing it to

☆This project was partially funded by MT-LAB, a VKR Centre of Excellence, and the Danish Strategic Research Council (project 2106-06-0028) “Aspects of Security for Citizens”.

“Corresponding address: DTU Informatics, Technical University of Denmark, Richard Petersens Plads Building 321, DK-2800 Kongens Lyngby, Denmark.

Email addresses: fy@imm.dtu.dk (Fan Yang), clh@imperial.ac.uk (Chris Hankin), nielson@imm.dtu.dk (Flemming Nielsen), riis@imm.dtu.dk (Hanne Riis Nielsen)

proceed. In concrete systems this is implemented as part of the operating system or as part of the interpreter for the language at hand (e.g. the Java byte code interpreter); in both cases as part of the trusted computing base. Sometimes it is found to be more cost effective to systematically modify the code so as to inline the checks that the reference monitor would otherwise have imposed [2]. In any case, even small modification in the security policies may involve substantial changes in the code or the underlying system.

The notion of aspect-oriented programming [3, 4] is an interesting approach to separation of crosscutting concerns. It is facilitated by modifications of existing languages; an example is AspectJ [3] that extends Java with so-called aspects. An aspect is an abstraction mechanism for encapsulating crosscutting concerns that goes beyond the traditional abstraction mechanisms such as classes and methods. A classical example of a crosscutting concern is logging as it affects every part of the program. The enforcement of other security policies is also an obvious candidate for such separation of crosscutting concerns, e.g. because the security policies can be implemented by more skilled or more trusted programmers, or indeed because security considerations can more easily be retrofitted to suit the (new) security policy. In many cases the aspect-oriented approach provides a more flexible way for dealing with modifications in security policies [5, 6, 7, 8, 9] than the use of reference monitors. Indeed, it facilitates the use of frameworks for security policies that may be well suited to the task at hand but that are perhaps not of general applicability and therefore not appropriate for incorporating into a reference monitor.

We shall follow the approach of AspectJ [3] and view an aspect as consisting of a pointcut and a body called an advice. The pointcut identifies so-called join points in the program being executed; the corresponding advice may then introduce additional or alternative behavior before the execution of the program proceeds; in the case of logging this additional behaviour will perform the necessary logging actions. The additional actions specified by the advice are thus injected into the original program and this process is called weaving. In this paper we are concerned with access control, that is, whether an operation performed by a subject on an object is permitted or denied. As we shall see in Section 3 the pointcuts of our aspects will therefore specify the potential operation being performed and the advice will tell whether it is permitted or denied – meaning that the “additional or alternative behaviour” that might be specified by the advice will be limited to either permitting the action or denying it.

In addition to enforcing traditional access control [1] we shall also be interested in enforcing policies for secondary use of data using predictive access control, that is, the access control depends on how the data obtained will be used in the future execution of the program [10]. Classical reference monitors on the other hand rely only on information gathered by monitoring execution steps [11], and perform history-based dynamic checks. However, security policies are often concerned with information flow that cannot be implemented correctly without a security check of the overall behavior of the program including the part that has not yet been executed. We shall show how our aspect-oriented approach can be used to intercept processes at the join points and in this way handle predictive access control.

Outline of the paper. We shall be based on the coordination language KLAIM [12] (reviewed in Section 2) that facilitates distribution of data, mobility of code and handling of dynamically evolving, open systems. Our main contribution is the design of AspectKLAIM, an aspect-oriented extension of KLAIM that facilitates the trapping of actions as well as processes, and the use
Table 1: KLAIM Syntax – Nets, Processes and Actions.

of AspectKLAIM for predictive access control. The syntax and semantics of AspectKLAIM is presented in Section 3. In Section 4 we illustrate how this language can be used to enforce not only traditional access control policies but also predictive access control policies. Section 5 evaluates our contribution. Finally, in Section 6 we present related work and we conclude in Section 7.

2. Background on KLAIM

KLAIM is a language specifically designed to program distributed systems consisting of several mobile components that interact through multiple distributed tuple spaces (or databases). KLAIM uses a Linda-like generative communication model [13] but, instead of using Linda’s global shared tuple space, KLAIM associates a local tuple space with each location of a net. Each location may also have processes associated with it; the KLAIM computing primitives allow programmers to distribute and retrieve data to and from locations of a net, to evaluate processes at remote locations and to introduce new locations to the net.

2.1. Syntax of KLAIM

The syntax of a fragment of KLAIM is displayed in Table 1. A net (in Net) is a parallel composition of located processes and located tuples. For simplicity, the components of a tuple can be location constants only. We use the notation \( \vec{l} \) to represent a sequence of location constants and we shall write \( \epsilon \) for the empty sequence.

A process (in Proc) can be a parallel composition of processes, a guarded sum of action prefixed processes, or a replicated process; the latter is indicated by the \( * \) operator. We shall write 0 for a nullary sum, \( a.P \) for a unary sum, and \( a_1.P_1 + a_2.P_2 \) for a binary sum. In examples we shall allow to dispense with trailing occurrences of 0.

An action (in Act) operates on locations, tuples and processes: a tuple can be output to a location, it can be input from a location and it can be read from a location; the difference between the latter

\[^1\] Compared with the original development of KLAIM, we do not allow processes to be components of tuples.
two is that the input action will delete the tuple being read whereas this is not the case for the read action. Furthermore, we have actions for spawning a process at a location and for creating a new location. The actual operations performed by the actions are called capabilities (in $\text{Cap}$).

We shall not distinguish between the real locations and the data$^2$: they will all be called locations (in $\text{Loc}$). We shall write $\ell$ for an entity that is either a location constant $l$ or an applied occurrence of a location variable $u$. A defining occurrence of location variable is written $!u$ and, as we shall see below, its scope will be the entire process to the right of the action in which it occurs.

Well-Formedness. We shall require that nets are closed meaning that all applied occurrences of location variables are within the scope of a defining occurrence. To express this we shall introduce the functions $bv$ and $fv$ for determining the bound and free variables of the sequences $\vec{\ell}$ of locations occurring in actions. The definitions are standard so we omit the details; as an example we have $bv(l, u, !v) = \{v\}$ and $fv(l, u, !v) = \{u\}$ as well as $bv(l, !u, u) = \{u\}$ and $fv(l, !u, u) = \{u\}$.

To simplify our development, we shall impose further well-formedness conditions on the $\text{in}$ and $\text{read}$ actions. An input action $\text{in}(\vec{\ell})@\ell$, and similarly a read action $\text{read}(\vec{\ell})@\ell$, is well-formed if the sequence $\vec{\ell} = \ell_1, \ldots, \ell_k$ (for $k \geq 0$) of locations is well-formed, and this is the case when the following two conditions are fulfilled:

- $\forall i, j \in \{1, \ldots, k\} : i \neq j \Rightarrow bv(\ell_i) \cap bv(\ell_j) = \emptyset$
- $bv(\vec{\ell}) \cap fv(\vec{\ell}) = \emptyset$

The first condition demands that a location variable cannot have multiple defining occurrences whereas the second condition prohibits it from occurring as a bound variable as well as a free variable in the same sequence. As an example we will disallow $\text{in}(!u, !u)@l$, $\text{in}(!u, u)@l$ as well as $\text{in}(u, !u)@l$.

2.2. Semantics of KCLAIM

Informally, the meaning of a KCLAIM program is as follows:

1. select a location for the next step of execution
2. if the process at the location is a guarded sum, then one of the enabled choices is chosen non-deterministically and executed as follows:
   - if the prefix is an $\text{out}$ action, then it can be performed
   - if the prefix is an $\text{in}$ or $\text{read}$ action then the action can only be performed if there is a matching tuple at the target location, and it will then result in the appropriate variables being bound in the continuation of the process
   - if the prefix is an $\text{eval}$ action then the process can be spawned at the target location

---

$^2$In an extension of this work one might employ a simple type system to make this distinction.
Table 2: KLAIM Reaction Semantics (on closed nets).

Table 3: KLAIM Structural Congruence (see text).

• if the prefix is a newloc action then the network is dynamically extended with a new location and the continuation process is given the address of that location

3. then return to Step 1

More formally, the semantics is given by a one-step reduction relation on nets and it is defined in Table 2. We make use of a structural congruence on nets; this is an associative and commutative (with respect to \(\parallel\)) equivalence relation and the interesting cases are defined in Table 3. The latter is quite standard and in the following we comment on the rules for the actions in Table 2.

The rule for out is rather straightforward\(^3\); it uses the fact that the action selected may be part of a guarded sum to dispense with any other alternatives. The rules for in and read only progress if the formal parameters \(\bar{\ell}\) match the candidate tuple \(\bar{l}\). The details of the matching operation are given in Table 4 (and explained below); if the matching succeeds and produces a substitution (denoted \(\theta\)) then the rule applies and we write \(P\theta\) for the continuation process where the formals are bound to the actual parameters; if no substitution is produced by the matching (due to a fail in part of the computation) then the rule does not apply. The rule for eval will spawn a new process at a specified location before continuing with the continuation process \(P\). Finally the rule for newloc will create a fresh empty location and substitute it for the formal parameter in the continuation process \(P\) (written \(P[l/u]\)).

\(^3\)The original semantics of KLAIM tests that the output location exists; we do not do so here as we rely on the invariant (maintained by newloc) that all locations mentioned in processes do in fact exist. Indeed this would interact nicely with the type system mentioned in the previous footnote.
\[
\text{match}(\ell, \ell'; l, l) = [l/\ell] \circ \text{match}(\ell'; l)
\]
\[
\text{match}(\epsilon; \epsilon) = \text{id}
\]
\[
\text{match}(\ldots; \ldots) = \text{fail}
\]

<table>
<thead>
<tr>
<th>Table 4: Matching Input Patterns to Data.</th>
</tr>
</thead>
</table>

The matching operation of Table 4 takes two parameters, the formals \(\ell\) and the actuals \(l\), and returns a substitution \(\theta\) being a (potentially empty) list of pairs of the form \([l/\ell]\); if the list is empty this is denoted by \(\text{id}\). As an example \(\text{match}(\ell, \ell'; l, l')\) returns \([l'/\ell]\) whereas \(\text{match}(\ell, \ell'; l, l')\) fails. Notice that the definition does not apply to location variables as the reaction semantics is restricted to closed nets; consequently the formals will only contain location constants and defining occurrences of variables and clearly the tuple space only contains location constants.

**Example 1.** To illustrate the semantics let us consider a simple Electronic Health Record system with a location EHDB containing tuples of the form \(\langle p, a, c \rangle\) where \(p\) is the name of the patient, \(a\) is the author of the medical record and \(c\) is the contents of the record. A user \(U\) may then read a medical record for a patient \(P\) and copy its contents to his/her own tuple space by the following process:

\[
U :: \text{read}(P, a, c)@\text{EHDB. out}(P, c)@U
\]

Let us assume that EHDB among others contains the tuple \(\langle P, A, C \rangle\). The matching required in the rule for read will succeed and produce the substitution \([A/a][C/c]\); once the action has been executed the continuation process will become \(U :: \text{out}(P, C)@U\). The output action can now be executed immediately and will place the tuple \(\langle P, C \rangle\) in the tuple space of \(U\). On the other hand if EHDB did not contain any tuples with \(P\) as first component then the matching of the read action would fail and the whole process would be stuck. \(\square\)

### 3. Introducing AspectKLAIM

The idea is now to extend KLAIM with a mechanism that, whenever an action is to be executed, will check whether this is permitted; the resulting language is called AspectKLAIM. The policies specifying when an action is permitted will be given by aspects and we shall assume that we have a fixed global set of such aspects governing the execution of our net. Each aspect will have a pointcut specifying which actions it may trap and it has an advice that will be evaluated when an action has been trapped. The evaluation of the advice will give break when certain conditions are met, and this will effectively postpone the execution of the thread, otherwise it will allow the thread to proceed. In contrast to AspectK [14], and aspect-oriented programming in general, we do not allow to modify the action being trapped, so in particular we cannot replace it by another action or just omit it.

Our aspects allow not only to trap actions but also to trap processes to be executed in the future. This can be a process that is to be evaluated at a remote site, or it can be the process continuation of a trapped action. In both cases it enables us to express predictive access control policies by analysing the actions of the trapped process. We shall elaborate on this in the next section.

---

6
3.1. Syntax of AspectKLAIM

The syntax of AspectKLAIM is an extension of Table 1 with the syntactic constructs of Table 5. Here we first introduce a system (in System) consisting of a net $N$ (in Net) and a sequence of global aspect declarations, written $\triangleright\triangleright asp$. An aspect declaration (in Asp) takes the form $A[cut; cond_A] \triangleq adv$: here $A$ is the name of the aspect, $cut$ specifies a pointcut for the action to be trapped, $cond_A$ is the applicability condition and $adv$ is the advice to be evaluated in case the action is trapped.

The syntax of pointcuts (in Cut) is similar to that of actions but additionally contains a process variable $X$ that will be bound to the continuation process. For the eval action we insist that the pointcut uses a process variable that then will be bound to the actual process being spawned and finally in the case of newloc we insist that the pointcut makes use of a $!u$ bound parameter.

Each advice (in Advice) gives a unique run-time suggestion about when to break and it otherwise allows the thread to proceed. It depends on the evaluation of a predictivity conditional $cond_P$. The predictivity condition ($cond_P$) is part of the novelty of AspectKLAIM whereas the applicability condition ($cond_A$) is familiar from aspect-oriented languages.

We support a variety of applicability conditional expressions (in BExpA). The primitive $\text{test}(@\ell | \tilde{c}) = true$ evaluates to true if there is a tuple in the tuple space of $\ell$ that matches $\tilde{c}$. In addition to basic boolean expressions, the condition $cond_A$ also includes membership tests on various sets and bounded existential quantification and universal quantification.

We also support a variety of predictivity conditional expressions (in BExpP). The primitives $\text{set}_1 = \text{set}_2$ and $\text{set}_1 \subseteq \text{set}_2$ allow for checking the equality or containment of two sets. Additionally we include the same basic boolean operations as for applicability conditionals.
The syntactic category **Set** of sets includes the usual set forming operators and is extended with special constants and operators that can be used to analyse processes for simple properties. Given a process, the operator \( \text{Loc}_c \) will return the set of target locations of all actions with capability \( c \); as an example the test \( \{ u \} \subseteq \text{Loc}_{\text{out}}(X) \) will check whether \( u \) is used as the target location of an output action in the process bound to \( X \). The operation \( \text{FV}_c \) will return the set of free location variables of all actions with capability \( c \) and the operation \( \text{LC}_c \) will return the set of location constants of all actions with capability \( c \). Finally we include the constant set \( \text{LVar} \), interpreted as the set of location variables occurring in the system of interest. It is possible to extend this part of the language with new analysis operators.

It is important to note that the meaning of occurrences of \( u \) and \( !u \) in a pointcut are slightly different from that of a KLAIM process. An occurrence of \( u \) can only match a constant location so the pointcut \( l_\varepsilon :: \text{in}(l, u)@l_0.X \) can be successfully matched against the process \( l_\varepsilon :: \text{in}(l, P)@l_0.P \) (and it will then result in binding \( P \) to \( u \)). On the other hand, an occurrence of \( !u \) in a pointcut can only be successfully matched against a defining occurrence of a location variable so the pointcut \( l_\varepsilon :: \text{in}(l, !u)@l_0.X \) can be successfully matched against the process \( l_\varepsilon :: \text{in}(l, !u')@l_0.P \) (and it will then bind \( u' \) to \( u \)). The details will be clarified when we explain the semantics of the language below.

**Well-formedness of Aspects.** Well-formedness of processes is defined as before so let us now consider the well-formedness conditions for aspects. We shall first define \( cl(cut) \) to be the list of entities involved in the pointcut; this is straightforward so we omit the details, some examples are \( cl(l_\varepsilon :: \text{in}(u, !u')@l_0.X) = \{ l_\varepsilon, u, !u', l_0, X \} \) and \( cl(l_\varepsilon :: \text{eval}(Y)@l_0.X) = \{ l_\varepsilon, Y, l_0, X \} \). The notion of well-formedness of sequences \( \xi^3 \) introduced in Section 2 is extended to sequences \( cl(cut) \) by additionally requiring that all occurrences of process variables are pairwise distinct.

A pointcut \( cut \) is then well-formed if \( cl(cut) \) is well-formed; as an example \( l_\varepsilon :: \text{eval}(Y)@l_0.X \) is well-formed whereas \( l_\varepsilon :: \text{eval}(X)@l_0.X \) is not. An applicability condition \( cond_\lambda \) is well-formed if each free location variable \( u \) is bound by some \( u \) occurring in \( cut \). A predictivity condition \( cond_P \) is well-formed if each free location variable \( u \) is bound by some \( !u \) occurring in \( cut \).

(Recall that a well-formed \( cut \) cannot contain both \( !u \) and \( u \) for the same variable \( u \).)

An aspect \( A[cut; cond_\lambda] \triangleq \textbf{break if } cond_P \) is well-formed if \( cut, cond_\lambda \) and \( cond_P \) are well-formed as explained above.

As an example the aspect \( A[l_\varepsilon :: \text{in}(l, !u)@l_0.X; -(u = b)] \triangleq \textbf{break if true} \) will not be well-formed as it makes use of \( u \) in the applicability condition. The rationale behind this decision is that although \( l_\varepsilon :: \text{in}(l, !u)@l_0.X \) can be successfully matched against a process as \( l_\varepsilon :: \text{in}(l, !u')@l_0.P \) it should not be possible to subsequently perform any tests on the actual value received.

In examples we allow the following shorthands: we may write \textbf{break} instead of \textbf{break if true}, and we may write \( \_ \) for any \( l, u, !u \) or \( X \) not used subsequently in the aspect.
\[
\text{let } \overrightarrow{as}\cdot p \text{ in } N \rightarrow \text{let } \overrightarrow{as}\cdot p \text{ in } N'
\]

\[
\overrightarrow{as}\cdot p \vdash N_1 \rightarrow \overrightarrow{as}\cdot p \vdash N'_1
\]

\[
\overrightarrow{as}\cdot p \vdash N_2 \rightarrow \overrightarrow{as}\cdot p \vdash N'_2 \parallel N_2
\]

\[
N = M \quad \overrightarrow{as}\cdot p \vdash M \rightarrow \overrightarrow{as}\cdot p \vdash M' \quad M' = N'
\]

\[
\overrightarrow{as}\cdot p \vdash N \rightarrow \overrightarrow{as}\cdot p \vdash N'
\]

Table 6: Reaction Semantics of AspectKLAIM (on closed nets).

3.2. Semantics

The semantics is given by a one-step reduction relation on well-formed systems, nets and actions. As before, we make use of the structural congruence on nets which is defined in Table 3. In addition, we also re-use the operation match in Table 4.

The reaction rules are defined in Table 6, where \( \overrightarrow{as}\cdot p \) is a global environment of aspects. The rules for actions all make use of the function \( \Phi \) for determining whether or not all applicable aspects allow the action to proceed (in which case it evaluates to true) or whether at least one aspect requires the action to break (in which case it evaluates to false). Subject to the definition of \( \Phi \) the rules of Table 6 are straightforward: if \( \Phi \) returns true then the action will be executed and gives the same result as in Table 2; otherwise the action and its continuation will be replaced by itself (denoted \( \text{LHS} \)) which effectively postpones the thread and avoids the creation of a covert channel. As in KLAIM, the action \( \text{out} \) simply puts the tuple \( \vec{t} \) into the location \( l_0 \) and continues with the continuation process \( P \); the actions \( \text{in} \) and \( \text{read} \) only progress if the formal parameters \( \vec{t} \) match those of a tuple \( \vec{t} \) in the relevant location, and this operation is defined in Table 4 as before. The \( \text{eval} \) action spawns a new process at the specified location before continuing with the continuation process. The \( \text{newloc} \) action creates a fresh empty location and proceeds with
The function $\Phi$ is defined in Table 7 and makes use of three auxiliary functions. The function $\text{extract}$ facilitates the checking procedure by producing a list of entities: the location where the trapped action is; the capability ($\text{out}$, $\text{in}$, $\text{read}$, $\text{eval}$ or $\text{newloc}$); the parameters of the action; the target location of the action; and the continuation process. The function is applied to pointcuts as well as join points; as an example $\text{extract}(l_{s} :: \text{in}(l, !u)@l_{0}.P) = (l_{s}, \text{in}, l, !u, l_{0}, P)$.

The function $\text{check}$, defined in Table 8, matches a pointcut against a join point and it produces the corresponding bindings of the parameters of the join point to those of the pointcut. The evaluation of the function $\text{check}$ relies on the evaluation of several invocations of $\text{do}$ that try to match every parameter in the pointcut against the corresponding parameter in the join point. If at least one mismatch occurs, $\text{check}$ will return $\text{fail}$. Notice that a variable $u$ in a pointcut can only match an actual location whereas a variable $!u$ can only match against binding occurrences of variables. As an example, the attempt to match a binding occurrence of variable $!u$ in a pointcut to an actual location $l$ from a join point will return $\text{fail}$.

Turning to the definition of $\Phi$ in Table 7 we notice that we inspect the aspects of $\text{−−→asp}$ one by one from left to right. If $\text{check}$ returns $\text{fail}$ it means that this aspect will not apply to the action and we shall continue by evaluating the next aspect against this action. If $\text{check}$ succeeds, it returns a substitution, $\theta$, which is applied to the applicability condition ($\text{cond}A$) as well as the predictivity condition ($\text{cond}P$), and $\text{check}$ will then continue searching through the remaining aspects, taking the conjunction of all results. Note that the order in which the aspects are listed in $\text{−−→asp}$ does not matter.

The conditions of the aspects are evaluated using the function $\llbracket \cdot \rrbracket$. It is implicit in the way it accesses the current tuple space to determine the value of expressions of the form $\text{test}(\vec{t})@\ell$. The substitution ($\theta$) obtained from $\text{check}$ will provide an applicability condition $\text{cond}A \theta$ without

<table>
<thead>
<tr>
<th>check($a; a'$)</th>
<th>$\text{do}(a; a') \circ \text{check}(\vec{a}; \vec{a'})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>check($\epsilon; \epsilon$)</td>
<td>$\text{id}$</td>
</tr>
<tr>
<td>check($_; _$)</td>
<td>fail otherwise</td>
</tr>
</tbody>
</table>

| $\text{do}(u; l)$ | $[l/u]$ |
| $\text{do}(!u; !u')$ | $[u'/u]$ |
| $\text{do}(l; l)$ | $\text{id}$ |
| $\text{do}(c; c)$ | $\text{id}$ |
| $\text{do}(X; P)$ | $[P/X]$ |
| $\text{do}(\_; \_)$ | fail otherwise |

Table 8: Checking Formals against Actuals.

---

The function $\Phi$ is defined in Table 7 and makes use of three auxiliary functions. The function $\text{extract}$ facilitates the checking procedure by producing a list of entities: the location where the trapped action is; the capability ($\text{out}$, $\text{in}$, $\text{read}$, $\text{eval}$ or $\text{newloc}$); the parameters of the action; the target location of the action; and the continuation process. The function is applied to pointcuts as well as join points; as an example $\text{extract}(l_{s} :: \text{in}(l, !u)@l_{0}.P) = (l_{s}, \text{in}, l, !u, l_{0}, P)$.
whereas we write \( \text{cap}(a) = c \) in the pointcut \( \text{cut} \); the latter are used in syntactic checks of the form \( \text{set}_1 = \text{set}_2 \) and \( \text{set}_1 \subseteq \text{set}_2 \) and remain uninterpreted.

Finally, the syntax of Table 5 introduces a number of basic operators for forming a syntactic analysis of the processes (usually the continuation of the trapped action bound by \( X \)). As already mentioned we write \( \text{LVar}_r \) for the set of location variables of the system of interest. The set-forming behavior analysis functions \( \text{Loc}_a \), \( \text{LC}_a \) and \( \text{FV}_a \) are formally defined in Table 9. As usual \( f v \) and \( b v \) return the set of free and bound variables of an action, respectively. Additionally we write \( \text{cap}(a) \) for the capability of the action \( a \), \( \text{loc}(a) \) for the target location of the action \( a \) (if any) and we write \( \text{lc}(a) \) for the location constants of the action \( a \) (possibly including the target locations). As an example we will have \( \text{Loc}_\text{in}(\text{in}(l, u, !v) @ l_0) = \{l_0\} \), \( \text{LC}_\text{in}(\text{in}(l, u, !v) @ l_0) = \{l, l_0\} \) whereas \( \text{FV}_\text{in}(\text{in}(l, u, !v) @ l_0) = \{u\} \).

In the next section we shall give examples of the use of these operators in enforcing predictive access control policies.

### 4. Traditional and Predictive Access Control

We first show how to use \text{AspectKLAIM} for expressing traditional access control (corresponding to the predictivity conditional \( \text{cond}_P \) always being \text{true}) and then show how to use \text{AspectKLAIM} for expressing predictive access control.

#### 4.1. Traditional Access Control

**Example 2.** Returning to Example 1 we shall now impose the policy that a user is only allowed to read a tuple from \( \text{EHDB} \) if he/she is a nurse or a doctor at the hospital or the user is the patient himself/herself. This can be expressed as follows:

\[
A_1[u :: \text{read}(p, \_ , \_ ) @ \text{EHDB} ; X ; \neg (\exists r \in \text{[nurse]} \cup \text{[doctor]} : \text{test}(u, r) @ \text{RDB} \lor p = u)]
\]

\( \triangleq \) \text{break}

Here we assume that \text{RDB} is a role database for the hospital containing pairs of staff members and their roles. We are using the don’t care pattern for the parameters of the \text{read} action we do
not want to constrain. The variable \( p \) is used to express further conditions in the advice as is the existentially quantified variable \( r \).

Consider now the process of Example 1

\[
U :: \text{read}(P, !a, !c)@\text{EHDB. out}(P, c)@U
\]

and assume that \( \langle U, \text{doctor} \rangle \) is in RDB. The matching of the pointcut against the \text{read} action will successfully produce the substitution \([U/u] \circ [P/p] \circ [\text{out}(P, c)@U/X] \) and turning to the evaluation of the applicability condition it will extend this substitution with \([\text{doctor}/r] \) showing that the applicability condition evaluates to false. Thus the first action of \( U \) can take place – and as we discussed in Example 1 it will only succeed if there is a tuple \( \langle P, A, C \rangle \) in EHDB with \( P \) as its first component and then the resulting process will be \( U :: \text{out}(P, C)@U \). As we currently have no aspects for the output actions this action can be executed immediately.

If neither \( \langle U, \text{doctor} \rangle \) nor \( \langle U, \text{nurse} \rangle \) occur in RDB then the applicability condition yields false if \( P=U \). On the other hand, if \( P \neq U \) then the applicability condition evaluates to true and hence the \text{read} action will not be permitted.

\textbf{Example 3.} Let us consider a variant of the above process where the name of the patient is left unspecified

\[
U' :: \text{read}(!p', !a, !c)@\text{EHDB. out}(p', c)@U'
\]

meaning that any tuple from EHDB might be read. Assume that we want to impose a policy prohibiting this from happening by enforcing that the patient’s name must be given explicitly. This can be done using the aspect:

\[
A_2[u :: \text{read}(p, _-_, c)@\text{EHDB. X}; \text{true}] \triangleq \text{break}
\]

To see this observe that the pointcut will match the \text{read} action of \( U' \) using the substitution \([U'/u] \circ [p'/p] \circ [\text{out}(p', c)@U'/X] \) and that the applicability condition trivially evaluates to true and consequently the \text{read} action of the process cannot take place.

It is worth pointing out that the use of \( !p \) in the pointcut of \( A_2 \) is crucial; as an alternative we might consider:

\[
A'_2[u :: \text{read}(p, _-_, c)@\text{EHDB. X}; \text{true}] \triangleq \text{break}
\]

The pointcut of this policy will not match the \text{read} action of \( U' \) as \( !p' \) cannot be matched with \( p \) (see Table 8). Therefore this aspect does not apply and hence the \text{read} action can proceed. Thus the aspect \( A'_2 \) is not adequate for capturing the intended policy.

\textbf{Example 4.} Let us assume that our system specifies both of the aspects \( A_1 \) and \( A_2 \) of Examples 2 and 3. In order for the \text{read} action of the process \( U \) to proceed both aspects have to permit it. As discussed in Example 2 the aspect \( A_1 \) will permit the \text{read} action if \( \langle U, \text{doctor} \rangle \) or \( \langle U, \text{nurse} \rangle \) is in RDB or if \( P = U \). The aspect \( A_2 \) will permit the \text{read} action in all cases; the observation being that the pointcut of \( A_2 \) fails to match the \text{read} action of \( U \) because \( P \) and \( !p \) do not match (see Table 8). Thus \( A_2 \) does not apply and therefore the progress of \( U \) is only governed by \( A_1 \).

Turning to the \text{read} action of the process \( U' \) we first observe that the aspect \( A_1 \) does not apply because the pointcut of \( A_1 \) cannot match the \text{read} action of \( U' \) as \( p \) and \( !p' \) do not match (see
Table 8). Thus $A_1$ does not prevent the read action from taking place. We have already seen in Example 3 that the aspect $A_2$ will deny the action. Thus despite the fact $A_1$ allowed the action the process $U'$ will not be able to proceed with the read action as $A_2$ denies it.

4.2. Predictive Access Control

We will now show how to use AspectKLAIM to enforce security policies that require behavior analyses of processes to be executed in the future, and as part of this, we shall illustrate how the set-forming behavior analysis functions (specified in Table 9) can be used to express policies for direct information flow. We shall consider two scenarios, one addressing remote execution of processes and another addressing continuation processes for actions.

In the previous examples we have been concerned with direct access control. In particular, we studied the process

$U :: \text{read}(P, !a, !c)@EHDB.out(P, c)@U$

and we imposed the restriction that the user is only allowed to execute the read action if he/she is a nurse or a doctor at the hospital or the user is the patient him/herself. The aspect $A_1$ introduced in Example 2 directly captured this requirement. However, we have not captured a potential indirect access. To see this consider the process

$U'' :: \text{eval}(U :: \text{read}(P, !a, !c)@EHDB.out(P, c)@U'')@P$

that simply spawns a process remotely on $P$ that will read a tuple in EHDB and write its first and third component in the tuple space of $U''$. If $A_1$ is the only aspect being enforced then there is nothing preventing this from happening. The following example illustrates how the primitives of AspectKLAIM can be used to impose various policies controlling such indirect access.

Example 5. One obvious possibility is to disallow anyone from performing eval actions on remote locations:

$A_3[u :: \text{eval}(Y)@v.X; \neg u = v] \triangleq \text{break}$

This is clearly very restrictive and we should like to be more permissive.

Our next choice is to allow the spawned process to execute any action as long as it is not a read action to EHDB.

$A_4[u :: \text{eval}(Y)@v.X; \neg u = v] \triangleq \text{break if } \{\text{EHDB}\} \subseteq \text{Loc}_{\text{read}}(Y)$

Here $\text{Loc}_{\text{read}}(Y)$ is the set of locations $\ell$ such that $\text{read}(...@\ell$ occurs in the process bound to $Y$ as can be seen from the general definition in Table 9. In the case of the process $U''$ the matching of the pointcut will return the substitution $[U''/u]@[P/v]@[\text{read}(P, !a, !c)@EHDB.out(P, c)@U'']/[0/X]$ and $\text{Loc}_{\text{read}}(\text{read}(P, !a, !c)@EHDB.out(P, c)@U'') = \{\text{EHDB}\}$ meaning that the eval action is not permitted.

Unfortunately, this is not enough as the user $U''$ can modify his actions to be

$U'' :: \text{out}(\text{EHDB})@U''.\text{read}(!db}@U''.\text{eval}(\text{read}(P, !a, !c)@db.out(P, c)@U'')@P$
It is easy to see that the aspect $A_5$ will now permit the `eval` action as the `read` action is not directly related to EHDB; we will have $L_{\text{Coread}}(\text{read}(P, !u, !c)@\text{out}(P, c)@U') = \{db\}$ so the predictivity condition of $A_5$ evaluates to `false`. To cater for such indirect access we can strengthen the aspect to be

$$A_5[u :: \text{eval}(Y)@X; \neg u = v] \triangleq \text{break if } \neg((\text{LVar}_* \cup \{\text{EHDB}\}) \cap \text{Loccoread}(Y) = \emptyset)$$

Here we use the set $\text{LVar}_*$ denoting the set of location variables in the system. The variable $db$ used in the revised process $U''$ will be in the set $\text{LVar}_*$ and hence $A_5$ will prevent the `eval` action from being performed.

In the following we shall be concerned with access control policies for secondary use of data. A classical example comes from the health care scenario where researchers are given access to medical records provided that the privacy and confidentiality of the patients are preserved meaning that their identity is not revealed. In such situations the access rights do not only depend on those of the users but also on how the data is used once access has been granted. We shall now see how AspectKCLAIM can be used to specify such policies by trapping the continuation process and analyse its behaviour before permitting the requested action.

Let us once again consider the process $U$ with the slight modification that the target of the `out` action is the tuple space $R$:

$$U_R :: \text{read}(P, !u, !c)@\text{EHDB} \cdot \text{out}(P, c)@R$$

If $U_R$ is, say, a researcher this is an example of secondary use of data and the output action is problematic as it mentions the identity $P$ of the patient and therefore we would like to deny the `read` action. However, if we change the process to

$$U_{R'} :: \text{read}(P, !u, !c)@\text{EHDB} \cdot \text{out}(U_{R'}, c)@R$$

then the `read` action should be allowed even in the case where $U_{R'}$ is a researcher.

**Example 6.** Let us first consider the aspect

$$A_6[u :: \text{read}(P, \_\_\_\_)@\text{EHDB} \cdot X; \text{true}] \triangleq \text{break if } \neg [p] \subseteq \text{Locout}(X)$$

The condition checks whether the identity of the patient $(p)$ mentioned in the medical record occurs in an output action in the continuation $(X)$ of the `read` action. To express this we use the set $\text{Locout}(X)$ defined in Table 9 to be the set of free location constants occurring in `out` actions in $X$. If $p$ is not used in this way then the `read` action can proceed.

The policy $A_6$ will allow $U_{R'}$ to proceed: the matching of the pointcut will give rise to the substitution $[U_{R'}/u]@\{P/p\}@\text{out}(U_{R'}, c)@R/X$ and since $L_{\text{Cout}}(\text{out}(U_{R'}, c)@R) = [U_{R'}, R]$ the test $[p] \subseteq [U_{R'}/R]$ evaluates to `false` and therefore the `read` action is permitted. On the other hand the `read` action of the process $U_R$ will be denied: the substitution is now $[U_R/u]@\{P/p\}@\text{out}(P, c)@R/X$ and since $L_{\text{Cout}}(\text{out}(P, c)@R)$ contains $P$ the predictivity condition evaluates to `true` and the action is denied independently of whom $U_R$ is.
We shall want to be more permissive and therefore we modify the aspect as follows:

\[
A_7[u :: \text{read}(p, d, \_@EHDB,X; -\text{test}(u, \text{doctor}@RDB \lor u = p)]
\triangleq \text{break if } [p] \subseteq \text{LC}_{\text{out}}(X)
\]

The applicability condition will allow doctors to perform the \text{read} action and similarly it will allow the patient him/herself to do so.

So far we have not imposed any restrictions on the target location \( R \) of the \text{out} action; we shall do this in the following aspect:

\[
A_8[u :: \text{read}(p, d, \_@EHDB,X; -\text{test}(u, \text{doctor}@RDB \lor u = p)]
\triangleq \text{break if } [p] \subseteq \text{LC}_{\text{out}}(X) \land \neg(\text{Loc}_{\text{out}}(X) \subseteq [d, p])
\]

As before the applicability condition restricts the aspects to the case where the user is neither a doctor nor the patient him/herself. If the identity of the patient occurs in an output action and at the same time there is an output to a location that neither is that of the patient nor that of the doctor nor the patient herself then the \text{read} action is denied. Here the test \((\text{Loc}_{\text{out}}(X) \subseteq [d, p])\) determines whether \( d \) and \( p \) are the only target locations of \text{out} actions and if that is not the case the \text{read} action must be denied.

Consider the process \( U_R \) above; the matching of the pointcut gives rise to the substitution \([U_R/u] \circ [P/p] \circ [a/d] \circ [\text{out}(P, c)@R/X] \) and \( \text{LC}_{\text{out}}(\text{out}(P, c)@R) = [P, R] \) so the first conjunct of the predictivity condition evaluates to \text{true}. The second conjunct also evaluates to \text{true} since we have \( \text{Loc}_{\text{out}}(\text{out}(P,c)@R) = [R] \) and \( \neg([R] \subseteq [a, P]) \) and therefore the \text{read} action will not be permitted. However, if we modify the process to be \( U_P \) so that the target location \( R \) is equal to \( P \) then the \text{read} action can proceed. The process \( U_P \) can still proceed as it does not output the identity of the patient. \( \square \)

**Example 7.** So far we only dealt with attempts to perform secondary use of medical records from explicitly named patients. We now consider the challenge of dealing with attempts to perform secondary use of medical records from arbitrary patients. First a researcher that forgets to exclude the personal data:

\[
U_{RR} :: \text{read}(p, l, a, l)@EHDB,\text{out}(p, c)@RR
\]

Next a researcher that remembers to exclude the personal data:

\[
U'_{RR} :: \text{read}(p, l, a, l)@EHDB,\text{out}(c)@RR
\]

A suitable aspect guarding against this situation might be:

\[
A_9[u :: \text{read}(q, \_@EHDB,X; \text{true})] \triangleq \text{break if } [q] \subseteq \text{FV}_{\text{out}}(X)
\]

Matching \( U_{RR} \) against \( A_9 \) we get the substitution \([U_{RR}/u] \circ [p/q] \circ [\text{out}(p, c)@RR/X] \) and the condition \([q] \subseteq \text{FV}_{\text{out}}(X) \) becomes \([p] \subseteq \text{FV}_{\text{out}}(\text{out}(p, c)@RR) \) so that the action of \( U_{RR} \) is denied. The action of \( U'_{RR} \) on the other hand will be permitted. \( \square \)
5. Evaluation

Access control policies address the situation where a subject asks to be allowed to perform a given access to a given object. In its simplest formulation this request is answered by a discretionary access control matrix that either grants or denies the request based on the subject, the access and the object. Very many extensions to this setup have been considered: mandatory access control, role based access control, time based access control, location based access control etc. as surveyed in for example [15]. They share the limitation that all decisions are based on the present state of the system (possibly extended with a record of the past history) but do not take the future into account: the access control decision is not based on the way in which data is actually going to be used.

This is particularly important in applications regarding the secondary use of data [10] where the noble aim of obtaining useful statistical information seems to override the usual considerations of privacy as encoded by the access control decisions. This may be overcome by allowing trusted subjects to bypass the default access control decisions, for example using the exceptions of [16]. But why should certain subjects be trusted?

The approach of this paper is to “de-mystify” the considerations of whom should be trusted by preventing the means for expressing trust in terms of how data is actually used. In other words, one can trust any subject that can demonstrate that data is only used in an acceptable manner. Whilst current discussions about a new European Data Protection Framework do not specifically address the secondary use issue, this kind of provision would seem to be essential if individual citizens are truly to be put in control of their personal data. Admittedly, the analysis presented in this paper is fairly modest in being based on a mere syntactic analysis of program text but more advanced analyses can be devised [17]. In this way, the only trusted components will be the security policies as embodied in the aspects.

This provides the flexibility needed to deal with secondary use of data as we have documented in our extended study of secondary use of data in health care [18]. It develops a large set of policies for an electronic health care system and we refer to that paper for a more thorough discussion of various design choices for expressing policies. The policies of [18] have been implemented in the proof-of-concept programming language AspectKE* [19], based on the core concepts of AspectKLAIM. The AspectKE* language can be compiled and executed under a Java environment for building secure distributed systems and is freely distributed4. The runtime system of AspectKE* is built on top of KJava [20], a Java package implementing the core concept of KLAIM. This work demonstrates that the policy language of AspectKLAIM presented in this paper can be efficiently implemented and executed. Besides the EHR system, [19] also presents a secure tuple space based chat application in AspectKE*. It is shown how to enforce access control policies which require analysis of future behavior of a process in a chat system that contains untrusted components. These two applications together demonstrate that the AspectKLAIM model can be efficiently implemented and executed in real world settings.

In our view, this approach opens up a complete new avenue to access control that extends well beyond the current study of aspect oriented access control in Turing-complete distributed coordi-

4http://www.graco.c.u-tokyo.ac.jp/ppp/projects/aspectklava.en
nation languages\(^5\). It allows for much more fine grained access control decisions where decisions can be based not only on the subject, the access and the object but also on the future use of data. We believe (but clearly cannot prove) that this flexibility is beyond what can adequately be captured by the more traditional approaches to access control.

6. Related Work

6.1. Policy Enforcement Mechanisms and Aspect-Oriented Programming

Inlined Reference Monitors (IRM) [2] use a load-time, trusted program rewriter to insert security code into a target application, resulting in a self-monitoring application that performs security checks as it executes. There are many IRM systems implemented by various program rewriters (e.g. [22, 23, 24, 25]), ensuring that different types of application will obey their corresponding security policies. Hamlen and Jones [26] propose an aspect-oriented security policy specification language SPoX for enforcement by IRMs which establish a formal connection between AOP and IRMs. JavaMOP [27] implements IRM by using AspectJ aspects as the instrumentation mechanism. AspectKLAAM takes the AOP approach to internalize the reference monitor for enforcing security policy to tuple space systems, and directly encodes security concerns in aspects.

Most research focuses on the class of security policies that can be enforced by monitoring execution of a target system [11] and hence are enforceable by traditional reference monitors. AspectKLAAM allows us to perform a behavior analysis on future execution of the target system, giving us the capability of enforcing policies that go beyond reference monitors. In [28], the authors outline several promising methods for enforcing security policies: IRM, type systems and certifying compilers. The authors also argue that synergies among these approaches will achieve remarkable results. We believe our approach – aspects with behavior analysis – is comparable as an alternative to IRM with type systems.

A number of AOP languages can identify the data-flow and control-flow between join points, which can serve as powerful policy enforcement mechanisms. AspectJ’s cflow [3] captures the control flow between join points. The dataflow pointcut [29] identifies join points based on the dataflow of information. Tracematches [30] can give advice based on the execution history of computation. However, these systems can only refer to the past and current events, in contrast AspectKLAAM can refer to future events. A few AOP languages propose mechanisms so that aspects can be triggered by control flow of a program in the future, e.g., pcf1ow [31] and transcut [32], however, they lack support for providing dataflow information about the future as AspectKLAAM does. Some advanced AOP languages (e.g., [33, 34]) offer ways of referring to the future behavior of a program in the aspect, which in theory can be used for specifying security policies that depend on the future control-flow and data-flow. However, as they usually lack formal semantics and also normally only offer access to low level (e.g., bytecode-level) information of a

\(^5\)AspectKLAAM is Turing-complete. A sub-language without aspects and restricted to a network with one node is essentially a Linda-like language. It can be shown that any Turing machine can be encoded in this sub-language; alternatively, a modest extension to the datatypes that are storable in tuples, allows us to follow the approach of Busi et al [21] and encode any RAM program.
program, this makes it hard to understand and develop appropriate underlying analyses for enforcing security policies. The formal semantics of AspectKLAIM clarifies the way of developing useful behavior (program) analyses and presenting the analysis results through appropriate language abstraction, and formally pave a way of integrating program analysis techniques into the policy specification and enforcement procedure.

6.2. Security With Aspect-Oriented Programming

In [35] the authors present general guidelines for how to compose access control aspects in AspectJ, while in [36] an enforcement of application-specific policies in an access control service is implemented in CaesarJ [37]. Phung and Sands [8] identify classes of reference monitor-style policies that can be defined and enforced by AspectJ and present a method to realize some history-dependent security policies which cannot be naturally expressed in AspectJ. Ramachandran et al. [38] discuss using AspectJ for implementing multilevel security and demonstrate how aspects, in comparison to traditional programming, can guarantee better security assurance. Similarly, AspectKLAIM can be used to enforce a wide range of security policies but focuses on access control.

Oliveira et al. [39] use their own rewrite-based system to express access control policies and then map them into an AspectJ program; in [40], availability requirements are expressed in a formal model that combines deontic and temporal logics, and are then translated into availability aspects in AspectJ. One advantage of these approaches, shared by AspectKLAIM, is that policies can be formalized through security oriented languages which are more suitable for security considerations than general purpose languages.

As we have done, some researchers design their own special purpose aspect languages or systems to study security enforcement mechanisms. For example, HarmlessAML [5] is an aspect-oriented extension of Standard ML, and has a type system that guarantees well-typed harmless advice does not interfere with the computation.

6.3. Distribution with Aspect-Oriented Programming

Much work has been done regarding how to deploy and weave aspects for distributed systems: some work is relevant for language design of distributed AOP with explicit distribution [41, 42, 43], other work explores the implementation of AOP middleware to support distributed AOP [44, 45, 46]. AspectKLAIM is closer to the language design of distributed AOP which naturally follows the KLAIM programming model and uses remote pointcuts [41] that identify join points in a program running on a different location. However, AspectKLAIM does not aim at enhancing the flexibility of mechanisms to deploy, instantiate and execute distributed aspects, e.g., support advice execution over remote hosts, as AWED [42] and ReflexD [43] have achieved, rather it focuses on integrating analysis components for reasoning about the local or mobile processes to support advanced access control in a distributed setting. Compared with these languages, AspectKLAIM provides a well defined security enforcement mechanism to tuple space systems that supports process mobility.

AO4BPEL [47] is an aspect extension of the process-oriented composition language BPEL, which was originally designed for composing Web Services. Work in [48, 49] discusses different prin-
ciples of using AOP to implement coordination systems (in AspectJ), but that are not related to security.

Recently, two variants of AspectK [14] have been proposed. In AspectKB [50] we show how to use Belnap Logic to deal with conflicts when distributed advices are composed in a coordination environment whereas in AspectKP [51] we introduce probabilities to describe the potential imperfections in enforcing the security policies of interest. Neither of these developments are able to deal with predictive access control.

6.4. Security in Coordination Languages and Security Policy Languages

Regarding policy enforcement in the KLAIM family of languages, some authors use control and data flow analyses that are written in the Flow Logic approach (e.g. [52, 53]), others use type systems (e.g. [54]), and [55] combine these two lines of work. They can be used to enforce very advanced security policies, however, all of them require the user to explicitly annotate policies in the main code (e.g. attach policies to each location), while our approach avoids this by specifying them inside the aspects, thus achieving a better separation of concerns.

Secure shared date-space coordination languages can be classified into two categories with regard to the underlying access control mechanisms [56]: the entity-driven approach (additional information, associated to resources such as tuple spaces, tuples and single data fields, list the entities which are allowed to access the resources) e.g. Secure Lime [57] and KLAIM [12]; and the knowledge-driven approach (resources are decorated with additional information and the processes can access the resources only in the event that they prove to keep the knowledge of such additional information) e.g., SecOS [58] and SecSpaces [59]. Our approach is suitable for expressing access control policies that fit both an entity-driven approach and a knowledge-driven approach, as the additional information is essentially expressed in aspects and is directly embedded in neither resources nor processes. Moreover, this additional information is not limited to the past and current facts used in the previous work but also facts about the future, e.g., how particular data will be used, which is useful for enforcing predictive access control policies.

Binder [60] and Cassandra [61] are very powerful logic-based security policy languages, which are both based on the datalog logic-programming language. There are other prominent policy languages like Protune [62], Rei [63], Ponder [64], and KeyNote [65], which can express basic access control policies very well. Only Ponder and Rei can express usage control through obligation policies but, unlike AspectKLAIM, neither language can enforce them and has to trust that the party receiving the data uses it in proper ways [66].

7. Conclusion

We have presented AspectKLAIM, an aspect-oriented extension of the coordination language KLAIM [67]. This has provided a concrete vehicle for presenting our approach; the distributed tuple spaces provide a natural model of the kind of system that motivated our work. However, the approach could equally well be applied to more classical process algebraic languages; the join points in this case being read and write accesses to channels.
Compared with our previous work on AspectK [14], AspectKLAIM empowers aspects to trap not only the matched actions but also the process continuation and the processes being evaluated remotely. AspectKLAIM also provides various behavior analysis functions and enables us to reason about the future execution of processes, which improves the capability of standard reference monitors that normally only deal with history based security policies. To achieve this, we simplify AspectK so that actions before and after the current action are not allowed. If we had allowed these actions, a safe behavior analysis would be very difficult to achieve, since the processes to be executed might execute more actions (inserted by aspects at runtime) than planned. This is an interesting direction for the future work and will require more powerful program analyses than the behavior analyses of the present paper.

Our behavior analyses perform checks on the syntactic form of process continuations. It is not difficult to imagine policies that require more sophisticated checks using semantics-based static analyses [68], for example to predict all possible data that can be stored in a certain tuple space [52, 53]. Such extensions remain as future work but, nevertheless, we find that the combination of aspects with behavior and static analysis techniques shows great potential for serving as a flexible and powerful mechanism for policy enforcement, and as a promising method of building security and trust in a distributed and mobile environment.

Acknowledgement

This project was partially funded by MT-LAB, a VKR Centre of Excellence, and the Danish Strategic Research Council (project 2106-06-0028) “Aspects of Security for Citizens”. We would like to thank the anonymous referees for their comments. Hidehiko Masuhara and Tomoyuki Aotani worked with us on AspectKE*; we appreciate their comments on the paper.

References

URL http://hal.inria.fr/inria-00071386/en/


