Advice for Coordination

Hankin, Chris; Nielson, Flemming; Nielson, Hanne Riis; Yang, Fan

Published in:
10th international conference on Coordination Models and Languages (Coordination'08)

Publication date:
2008

Document Version
Early version, also known as pre-print

Advice for Coordination

Chris Hankin[1], Flemming Nielson[2], Hanne Riis Nielson[2], and Fan Yang[2]

[1] Department of Computing, Imperial College London
clh@imperial.ac.uk
[2] Department of Informatics, Technical University of Denmark
{nielson,riis,fy}@imm.dtu.dk

Abstract. We show how to extend a coordination language with support for aspect oriented programming. The main challenge is how to properly deal with the trapping of actions *before* the actual data have been bound to the formal parameters. This necessitates dealing with *open* joinpoints – which is more demanding than the closed joinpoints in more traditional aspect oriented languages like AspectJ. The usefulness of our approach is demonstrated by mechanisms for discretionary and mandatory access control policies, as usually expressed by reference monitors, as well as mechanisms for logging actions.

1 Introduction

Motivation. Software development faces the challenge of guaranteeing the compliance of software to security policies even when the software has been developed without adequate considerations of security. This situation might arise due to lack of skills of the application programmers, due to lack of trust in the application programmers or even due to modifications of the security properties after the original development of the sofware (e.g. to cater for new needs of the users).

Taking access control as an example, a number of schemes for discretionary access control (e.g. based on capability lists or access control lists) and mandatory access control (e.g. the Bell LaPadula policy for confidentiality) have been proposed for controlling the execution of software [13]. As an example, the attempt to read from a file where the program has insufficient access rights should not be successful. As another example, transferring data from a file with high security classification to a file with low security classification should also not be successful.

The traditional approach to enforcing such security policies is through a *reference monitor* [13] 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 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. When modelled using operational semantics, a reference monitor is usually a side condition to an inference rule either preventing or allowing the rule to be applicable. Sometimes it is found to be more cost effective to systematically modify the code so as to explicitly perform the checks that the reference monitor would otherwise have imposed; the term *inlined reference monitors* [9] has been coined for this.

An interesting approach to separation of concerns when programming systems is presented by the notion of *aspect oriented programming* [15, 16]. The enforcement of security policies is an obvious candidate for such separation of concerns, e.g. because the security policy can be implemented by more skilled or more trusted programmers, or indeed because security considerations can be retrofitted by (re)defining *advice* to suit the (new) security policy. This requires that a notion of aspects is supported by the programming language. The detailed definition of the advice will then make decisions about how to possibly modify the operation being trapped. In concrete systems this calls for a modified language (like AspectJ [3] for Java) that supports the use of aspects. When modelled using operational semantics a notion of trapping operations and applying advice needs to be incorporated. Usually, it is found to be more cost effective to systematically modify the code so as to explicitly perform the operations that the advice would otherwise have imposed; the term *weaving* (e.g. [3]) has been coined for this.

In many cases the aspect oriented approach provides a more flexible way to deal with modifications in security policies [8, 10, 11, 21, 24] than the use of reference monitors [20]. It facilitates to use 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 should like to refer to this process as *internalising* the reference monitor into a piece of advice. An example would be the enforcement of policies related to information flow or policies targeting the explicit needs of individuals; in particular this applies to the modelling of discretionary and mandatory access control policies [13] as well as mechanisms for logging actions. Since we do not offer priorities on advice we shall assume that the provision of advices is a priviledged operation.

Contribution. Our main contribution is the integration of aspects into a coordination language that facilitates distribution of data, mobility of code, and the ability to work with dynamically evolving, open systems. Rather than invent a completely new language, we define a small kernel language for mobile agents based on KLAIM [5, 18, 19]. We present this language in Section 2; as in KLAIM, processes and action prefixes (LINDA's **read**, **in**, and **out**) are located.

In our extension action prefixes are the potential joinpoints – the places where execution can be interrupted by a piece of advice. We take the approach that input actions should be trapped *before* a concrete tuple has been selected for input. This is because we find that the alternative approach, to trap after a concrete tuple has been selected for input, would constitute a covert channel [12, 13]; indeed, the presence or absence of a tuple in the tuple space might either enable or prevent the advice to trap the action and this would amount to visible behaviour bypassing the security policy.

Trapping an input action *before* a concrete tuple has been selected for input requires our ability to deal with joinpoints that contain constructs for *binding* new variables – we shall call these *open* joinpoints. This is considerably more challenging than the closed joinpoints of traditional aspect oriented language like AspectJ [15]. To be more concrete, when we trap a method call in AspectJ we trap the actual call, i.e. the method name with its actual parameters, rather than the definition of the method, i.e. the method name with its *formal* parameters; in other words AspectJ traps closed joinpoints rather than *open* joinpoints. We show how to solve this challenge in Section 3 and provide a series of examples in Section 4.

The design space for how to introduce advice into coordination languages is quite broad. We have aimed for a modest approach being inspired by the operations of reference monitors; they generally allow to block an action or to let it proceed. A number of extensions can be foreseen – some of these are rather straightforward whereas others pose considerable difficulties; as a case in point it is nontrivial to add advice for ignoring or redirecting a given action. We discuss parts of the design space in Section 5.

2 KLAIM

The syntax of our fragment of KLAIM is defined in Table 1. We restrict ourselves to a core language for presentational purposes; it is straightforward to add the actions **newloc** and **eval** but we will not need these for the examples. Despite the rather modest selection of operations in our language it is still useful for quite a variety of applications related to business processes and similar workflow applications.

A net N is a parallel composition of located processes or located tuples. For simplicity, components of tuples can be location constants only. Nets must be closed, meaning that all variables must be in scope of a defining occurrence.

A process P is a parallel composition of processes, a guarded sum of action prefixed processes, or a replication (indicated by the $*$ operator). The guarded sum $\sum_i a_i.P_i$ is written 0 if the index set is empty.

$N \in \mathbf{Net}$	$N ::= N_1 \,\|\|\, N_2 \;\mid\; l :: P \;\mid\; l :: \langle\, \overrightarrow{l} \,\rangle$	
$P \in \mathbf{Proc}$	$P ::= P_1 \mid P_2 \;\mid \sum_i a_i.P_i \;\mid\; *P$	
$a \in \mathbf{Act}$	$a ::= \mathbf{out}(\overrightarrow{\ell})@\ell \mid \mathbf{in}(\overrightarrow{\ell^\lambda})@\ell \mid \mathbf{read}(\overrightarrow{\ell^\lambda})@\ell$	
$\ell, \ell^\lambda \in \mathbf{Loc}$	$\ell ::= u \;\mid\; l$	$\ell^\lambda ::= \ell \mid !u$

Table 1. KLAIM Nets and Processes Syntax

A tuple can be output to a location, input from a location, or read from a location (meaning that it is not removed). Parameters can be location constants l, defining occurrences of location variables $!u$, and applied occurrences of a location variable u. We use ℓ for location expressions (i.e. location variables and constants); and in patterns we use ℓ^λ which in addition to location expressions also include defining occurrences of locations. The scope of a defining occurrence is the entire process to the right of the occurrence.

Example 1. Assume that the location YP, for Yellow Pages, contains pairs of values representing names and phone numbers and that the location DB, for the database of a phone company, contains triples of values representing a particular phone call, that is, the phone number of the caller, the cost of the call and the name of the recipient. Consider the process:

$$\begin{aligned} \mathsf{User} :: \; & \mathbf{read}(!name, !telno)@\mathsf{YP}. \\ & \mathbf{read}(telno, !val_1, !val_2)@\mathsf{DB}. \\ & \mathbf{out}(val_1)@name \end{aligned}$$

Here User will first read a pair from the location YP and assign its two components to the variables *name* and *telno*. Next the location DB is consulted to read a triple whose first component equals the value of *telno* and the corresponding second component is assigned to the variable val_1 and the corresponding third component is assigned to the variable val_2. The final construct will write the first value to the location associated with *name*.

Well-formedness of Locations and Actions. To express the well-formedness conditions we shall introduce functions *bv* and *fv* for calculating the bound, resp. free, variables of the various kinds of locations that may occur in actions. The definitions are standard, in particular, $bv(l, u, !v) = \{v\}$ whereas $fv(l, u, !v) = \{u\}$.

An input action is well-formed if its sequence $\overrightarrow{\ell^\lambda} = \ell_1, \cdots, \ell_k$ (for $k \geq 0$) of locations is well-formed and this is the case when the following two conditions are fulfilled:

$$\begin{aligned} & \forall i, j \in \{1, \cdots, k\} : i \neq j \Rightarrow bv(\ell_i^\lambda) \cap bv(\ell_j^\lambda) = \emptyset \text{ and} \\ & bv(\overrightarrow{\ell^\lambda}) \cap fv(\overrightarrow{\ell^\lambda}) = \emptyset \end{aligned}$$

$$l :: P_1 \mid P_2 \equiv l :: P_1 \,||\, l :: P_2 \qquad\qquad l :: *P \equiv l :: P \mid *P$$

$$\frac{N_1 \equiv N_2}{N \,||\, N_1 \equiv N \,||\, N_2}$$

Table 2. KLAIM Structural Congruence.

$$l_s :: \mathbf{out}(\overrightarrow{l})@l_0.P + \cdots \rightarrow l_s :: P \,||\, l_0 :: \langle \overrightarrow{l} \rangle$$

$$l_s :: \mathbf{in}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \,||\, l_0 :: \langle \overrightarrow{l} \rangle \rightarrow l_s :: P\theta \qquad \text{if } match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$$

$$l_s :: \mathbf{read}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \,||\, l_0 :: \langle \overrightarrow{l} \rangle \rightarrow l_s :: P\theta \,||\, l_0 :: \langle \overrightarrow{l} \rangle \text{ if } match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$$

$$\frac{N_1 \rightarrow N_1'}{N_1 \,||\, N_2 \rightarrow N_1' \,||\, N_2} \qquad \frac{N \equiv N' \quad N' \rightarrow N'' \quad N'' \equiv N'''}{N \rightarrow N'''}$$

Table 3. KLAIM Reaction Semantics (on closed nets).

The first condition demands that we do not use multiple defining occurrences of the same variable in an action. The second condition requires that bound variables and free variables cannot share any name in a single action. Thus we shall disallow $\mathbf{in}(!u, u)@l$ as well as $\mathbf{in}(!u, !u)@l$.

Semantics of KLAIM. The semantics is given by a one-step reduction relation on nets and is defined in Table 3. We make use of the structural congruence on nets; this is an associative and commutative (with respect to $||$) equivalence relation and the interesting cases are defined in Table 2.

The rule for **out** is rather straightforward; 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 $\overrightarrow{\ell^\lambda}$ match the candidate tuple $\overrightarrow{l}$. The details of the matching operation are given in Table 4 (explained below); if the matching succeeds and produces a substution then the rule applies; if no substitution is produced (due to a fail in part of the computation) the rule does not apply.

The matching operation of Table 4 returns a substitution θ being a (potentially empty) list of pairs of the form $[l/u]$; if the list is empty it is denoted by id. Notice that the definition does not treat location variables because tuples in the tuple space may only contain location constants and the reaction semantics is restricted to closed nets.

$$match(\langle\rangle;\langle\rangle) = id$$

$$match(\langle \ell_1'^{\lambda}, \cdots, \ell_k'^{\lambda}\rangle;\langle l_1, \cdots, l_k\rangle) = \text{let } \theta = \text{case } \ell_1'^{\lambda} \text{of}$$
$$l_1' : \quad \text{if } l_1' = l_1 \text{ then } id \text{ else } \mathsf{fail}$$
$$!u : \quad [l_1/u]$$
$$\text{in } \theta \circ match(\langle \ell_2'^{\lambda}, \cdots, \ell_k'^{\lambda}\rangle;\langle l_2, \cdots, l_k\rangle)$$

Table 4. KLAIM Pattern Matching of Templates against Tuples.

Example 2. Continuing Example 1 we may consider the following net and some steps of its execution:

$$\begin{array}{l}
\quad \mathsf{YP} :: \langle \mathsf{Alice}, 55010\rangle \;||\; \mathsf{YP} :: \langle \mathsf{Bob}, 58266\rangle \\
||\;\; \mathsf{DB} :: \langle 55010, 100, \mathsf{Bob}\rangle \;||\; \mathsf{DB} :: \langle 58266, 1000, \mathsf{Alice}\rangle \\
||\;\; \mathsf{User} :: * \,\mathbf{read}(!name, !telno)@\mathsf{YP}.\; \mathbf{read}(telno, !val_1, !val_2)@\mathsf{DB}.\; \mathbf{out}(val_1)@name \\
\rightarrow \\
\quad \mathsf{YP} :: \langle \mathsf{Alice}, 55010\rangle \;||\; \mathsf{YP} :: \langle \mathsf{Bob}, 58266\rangle \\
||\;\; \mathsf{DB} :: \langle 55010, 100, \mathsf{Bob}\rangle \;||\; \mathsf{DB} :: \langle 58266, 1000, \mathsf{Alice}\rangle \\
||\;\; \mathsf{User} :: \mathbf{read}(55010, !val_1, !val_2)@\mathsf{DB}.\; \mathbf{out}(val_1)@\mathsf{Alice} \\
\qquad\quad | * \,\mathbf{read}(!name, !telno)@\mathsf{YP}.\; \mathbf{read}(telno, !val_1, !val_2)@\mathsf{DB}.\; \mathbf{out}(val_1)@name \\
\rightarrow \\
\quad \mathsf{YP} :: \langle \mathsf{Alice}, 55010\rangle \;||\; \mathsf{YP} :: \langle \mathsf{Bob}, 58266\rangle \\
||\;\; \mathsf{DB} :: \langle 55010, 100, \mathsf{Bob}\rangle \;||\; \mathsf{DB} :: \langle 58266, 1000, \mathsf{Alice}\rangle \\
||\;\; \mathsf{User} :: \mathbf{out}(100)@\mathsf{Alice} \\
\qquad\quad | * \,\mathbf{read}(!name, !telno)@\mathsf{YP}.\; \mathbf{read}(telno, !val_1, !val_2)@\mathsf{DB}.\; \mathbf{out}(val_1)@name \\
\rightarrow \\
\quad \mathsf{YP} :: \langle \mathsf{Alice}, 55010\rangle \;||\; \mathsf{YP} :: \langle \mathsf{Bob}, 58266\rangle \\
||\;\; \mathsf{DB} :: \langle 55010, 100, \mathsf{Bob}\rangle \;||\; \mathsf{DB} :: \langle 58266, 1000, \mathsf{Alice}\rangle \\
||\;\; \mathsf{Alice} :: \langle 100\rangle \\
||\;\; \mathsf{User} :: * \,\mathbf{read}(!name, !telno)@\mathsf{YP}.\; \mathbf{read}(telno, !val_1, !val_2)@\mathsf{DB}.\; \mathbf{out}(val_1)@name
\end{array}$$

In the first step User spawns a thread and reads the pair $\langle \mathsf{Alice}, 55010\rangle$ from YP; the bindings of the variables *name* and *telno* are reflected in the continuation of the thread. In the second step it is only possible to read a triple from DB that has 55010 as its first component; this results in binding val_1 and val_2 to 100 and Bob, respectively. The last step will then complete the thread by outputting the value 100 to Alice.

Example 3. Returning to Example 1 we may want to impose the condition that only some users are allowed to access the location DB containing secret data whereas all users are allowed to read from the location YP containing only public data. This can be expressed with discretionary access control using an *access control matrix* DAC containing triples (s, o, a) identifying which subjects s can perform which operations a on which objects o. We may thus equip the semantics of KLAIM with a reference monitor that will consult DAC whenever an action is executed; in particular, whenever User is performing a **read** action on a location

$S \in \mathbf{System}$	S	$::= \mathbf{let}\ \overrightarrow{asp}\ \mathbf{in}\ N$
$asp \in \mathbf{Asp}$	asp	$::= A[cut] \triangleq body$
$body \in \mathbf{Advice}$	$body$	$::= \mathbf{case}\ (cond)\ sbody\ ;\ body \mid sbody$
	$sbody$	$::= as\ \mathbf{break} \mid as\ \mathbf{proceed}\ as$
$as \in \mathbf{Act}^*$	as	$::= a.as \mid \varepsilon$
$cond \in \mathbf{BExp}$	$cond$	$::= \mathbf{test}(\overrightarrow{\ell^\lambda})@\ell \mid \ell_1 = \ell_2 \mid cond_1 \wedge cond_2 \mid \neg\ cond$
$cut \in \mathbf{Cut}$	cut	$::= \ell :: a$
$\ell^\lambda \in \mathbf{Loc}$	ℓ^λ	$::= \ell \mid !u \mid ?u$

Table 5. AspectK Syntax

l it will check whether $(\mathsf{User}, l, \mathbf{read}) \in \mathsf{DAC}$ and only proceed if this is the case. Similarly when performing an **out** action on some location l it will check whether $(\mathsf{User}, l, \mathbf{out}) \in \mathsf{DAC}$ before proceeding.

A comparable policy can be imposed by a reference monitor based on mandatory access control. Here *security levels* are assigned to subjects and object. In the simple case of just two security levels we may give DB the level high and YP the level low. The Bell-LaPadula security policy will then impose that a low user can only perform **read** actions on YP whereas **out** actions can be performed on any location. A high user, on the other hand, will be able to perform **read** actions on both YP and DB. The out action can only be performed on high locations unless a notion of declassification is imposed that will lower the users' security level.

3 AspectK

Syntax. The syntax of AspectK extends the syntax of KLAIM (Table 1) as shown in Table 5. A system S consists of a net N prefixed by a sequence of aspect declarations. An aspect declaration takes the form $A[cut] \triangleq body$, where A is the name of the aspect, *cut* is the action to be trapped by A and *body* specifies the way it should be handled.

The keyword **break** indicates that the original action is suppressed and prevents the process from being further executed, whereas the keyword **proceed** allows the original action to execute. In case of multiple aspects that trap an action, all the before actions are executed in declaration order, then the original action (in case of no **break**), and finally the after actions in reverse declaration order. The keyword **break** takes precedence over the keyword **proceed**.

$$\frac{N \to N' \quad \text{(where globally } \Gamma_A = \overrightarrow{asp})}{\textbf{let } \overrightarrow{asp} \textbf{ in } N \to \textbf{let } \overrightarrow{asp} \textbf{ in } N'}$$

$$l_s :: \underline{\textbf{stop}}.P + \cdots \to l_s :: 0$$

$$l_s :: \underline{\textbf{out}}(\overrightarrow{l})@l_0.P + \cdots \to l_s :: P \mid\mid l_0 :: \langle \overrightarrow{l} \rangle$$

$$l_s :: \underline{\textbf{in}}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta \qquad \text{if } match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$$

$$l_s :: \underline{\textbf{read}}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \text{ if } match(\overrightarrow{\ell^\lambda}; \overrightarrow{l}) = \theta$$

$$\frac{l_s :: \Phi_{\textbf{proceed}}(\Gamma_A; l_s :: \textbf{out}(\overrightarrow{l})@l_0).P \to N}{l_s :: \textbf{out}(\overrightarrow{l})@l_0.P + \cdots \to N}$$

$$\frac{l_s :: \Phi_{\textbf{proceed}}(\Gamma_A; l_s :: \textbf{in}(\overrightarrow{\ell^\lambda})@l_0).P \mid\mid N' \to N}{l_s :: \textbf{in}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \mid\mid N' \to N}$$

$$\frac{l_s :: \Phi_{\textbf{proceed}}(\Gamma_A; l_s :: \textbf{read}(\overrightarrow{\ell^\lambda})@l_0).P \mid\mid N' \to N}{l_s :: \textbf{read}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \mid\mid N' \to N}$$

Table 6. Reaction Semantics (on closed nets)

The *cond* is similar to a standard boolean expression, which will be evaluated to $true$ or $false$. The primitive $\textbf{test}(\overrightarrow{\ell^\lambda})@\ell$ will only be evaluated to $true$ in case that there is a tuple that matches $\overrightarrow{\ell^\lambda}$ in the tuple space at location ℓ.

A cutpoint *cut* is simply a cut action accompanied by location ℓ. For the use in cut actions we have extended the definition of ℓ^λ to incorporate a new location expression $?u$ that is intended to trap both $!u$ and l occurring in actions; this will be made precise in the definition of the *check* function in Table 9.

Well-formedness of Cuts. We define *cl*(*cut*) that generates a list of entities involved in a cut. For example:

$$cl(l_s :: \textbf{in}(!x, y, ?z)@l_0) = \langle l_s, x, y, z, l_0 \rangle$$

In addition to the well-formedness conditions for KLAIM, we require that the variables of *cl*(*cut*) are pairwise distinct. When $!u$ or $?u$ is used in a cut pattern, u should only occur in the *after* actions (actions that occur after **proceed**); in particular u should neither be used in any before action nor in any conditionals. No use of $?u$ will be allowed inside tests (use $!u$ instead).

Semantics of AspectK. The semantics is given by a one-step reduction relation on well-formed systems and nets. The interesting rules are defined in Table 6 – the rule for reduction on nets and a congruence rule (see Table 3) are omitted – and we also make use of the structural congruence on nets defined in Table 2.

The rules for the three actions come in pairs, as is illustrated in Table 6. One rule takes care of the action when no advice is allowed to interrupt it; this is syntactically denoted by underlining.

$$\begin{array}{rcl}
\Phi_f(A[cut] \triangleq body, \Gamma_A; \ell :: a) = \text{case } trap(cut, \ell :: a) \text{ of } \mathsf{fail}: & & \Phi_f(\Gamma_A; \ell :: a) \\
& \theta: & \kappa_f^{\Gamma_A, \ell::a}(body\ \theta) \\
\Phi_f(\varepsilon; \ell :: a) = \text{case } f \text{ of } \mathbf{proceed}: \underline{a} & & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{break}: \underline{\mathbf{stop}} & &
\end{array}$$

Table 7. Trapping Aspects: Step 1.

$$\begin{array}{rl}
trap(cut, \ell :: a) = \text{case } (cut, \ell :: a) \text{ of} & \\
\quad (\ell_s :: \mathbf{out}(\overrightarrow{\ell})@\ell_0, l_s :: \mathbf{out}(\overrightarrow{l})@l_0): & check(\langle \ell_s, \overrightarrow{\ell}, \ell_0\rangle, \langle l_s, \overrightarrow{l}, l_0\rangle) \\
\quad (\ell_s :: \mathbf{in}(\overrightarrow{\ell^\lambda})@\ell_0, l_s :: \mathbf{in}(\overrightarrow{\ell'^\lambda})@l_0): & check(\langle \ell_s, \overrightarrow{\ell^\lambda}, \ell_0\rangle, \langle l_s, \overrightarrow{\ell'^\lambda}, l_0\rangle) \\
\quad (\ell_s :: \mathbf{read}(\overrightarrow{\ell^\lambda})@\ell_0, l_s :: \mathbf{read}(\overrightarrow{\ell'^\lambda})@l_0): & check(\langle \ell_s, \overrightarrow{\ell^\lambda}, \ell_0\rangle, \langle l_s, \overrightarrow{\ell'^\lambda}, l_0\rangle) \\
\quad \text{otherwise } \mathsf{fail} &
\end{array}$$

Table 8. Trapping Aspects: Step 2.

$$\begin{array}{rl}
check(\langle\rangle, \langle\rangle) = id & \\
check(\langle \ell_1^\lambda, \ell_2^\lambda, \cdots, \ell_k^\lambda\rangle, \langle \ell_1'^\lambda, \cdots, \ell_k'^\lambda\rangle) = \text{let } \theta = & \text{case } (\ell_1^\lambda, \ell_1'^\lambda) \text{ of} \\
& \quad (!u, !u'): [u'/u] \\
& \quad (?u, !u'): [u'/u] \\
& \quad (?u, l'): [l'/u] \\
& \quad (u, l'): [l'/u] \\
& \quad (l, l'): \text{if } l = l' \text{ then } id \text{ else } \mathsf{fail} \\
& \text{otherwise } \mathsf{fail} \\
\text{in } \theta \circ check(\langle \ell_2^\lambda, \cdots, \ell_k^\lambda\rangle, \langle \ell_2'^\lambda, \cdots, \ell_k'^\lambda\rangle) &
\end{array}$$

Table 9. Trapping Aspects: Step 3.

The rules for the non-underlined actions all take the same shape and make use of the function Φ defined in Table 7. The result of $\Phi_f(\Gamma_A; \ell :: a)$ is a sequence of actions trapping $\ell :: a$; Γ_A is a global environment of aspects. The index f is either **proceed** or **break**. In general f will be **break** if at least one "**break**" advice applies, otherwise it will be **proceed**. In case of **proceed** the action $\underline{a}$ is eventually emitted, otherwise it will be dispensed with and be replaced with the **stop** action, killing all the subsequent actions. Recall that advice is searched in the order of declaration and applies in a parenthesis-like fashion.

The function Φ uses an auxiliary function *trap* (see Table 8) to step through the aspects in the aspects environment. In each case, *trap* checks whether the

$$\kappa_f^{\Gamma_A,\ell::a}(\textbf{case}\ cond\ sbody\ ;\ body) = \text{case}\ B(cond)\ \text{of}\ \begin{array}{rl}\textbf{tt}: & \kappa_f^{\Gamma_A,\ell::a}(sbody)\\ \textbf{ff}: & \kappa_f^{\Gamma_A,\ell::a}(body)\end{array}$$

$$\kappa_f^{\Gamma_A,\ell::a}(sbody) = \text{case}\ sbody\ \text{of}\ as_1\ \textbf{proceed}\ as_2 : as_1.\Phi_f(\Gamma_A;\ell::a).as_2\ \ as\ \textbf{break}\ : as.\Phi_{\textbf{break}}(\Gamma_A;\ell::a)$$

Table 10. Trapping Aspects: Step 4.

$$B(\textbf{test}(\overrightarrow{\ell^\lambda})@l) = \begin{cases}\textbf{tt}\ \text{if there exists a tuple}\ \overrightarrow{l}\ \text{at location}\ l\\ \quad\text{such that}\ match(\overrightarrow{\ell^\lambda};\overrightarrow{l}) \neq \mathsf{fail}\\ \textbf{ff}\ \text{otherwise}\end{cases}$$

$$B(l_1 = l_2) = \begin{cases}\textbf{tt}\ \text{if}\ l_1 = l_2\\ \textbf{ff}\ \text{otherwise}\end{cases}$$

$$B(cond_1 \wedge cond_2) = \begin{cases}\textbf{tt}\ \text{if}\ B(cond_1) = \textbf{tt}\ \text{and}\ B(cond_2) = \textbf{tt}\\ \textbf{ff}\ \text{if}\ B(cond_1) = \textbf{ff}\ \text{or}\ B(cond_2) = \textbf{ff}\end{cases}$$

$$B(\neg cond) = \begin{cases}\textbf{tt}\ \text{if}\ B(cond) = \textbf{ff}\\ \textbf{ff}\ \text{if}\ B(cond) = \textbf{tt}\end{cases}$$

Table 11. Trapping Aspects: Step 5.

cut matches the action; the check is accomplished by using a further auxiliary function, *check* (see Table 9), which either fails or produces a substitution for the variables occurring in the cut. The *check* function is essentially an extension of the *match* function (see Table 4) to accommodate the matching of cut patterns. If a cut matches a normal action, we use $\kappa_f^{\Gamma_A,\ell::a}$ (see Table 10) to recursively search for further advices; *body* θ is computed in the obvious way.

The $\kappa_f^{\Gamma_A,\ell::a}$ function processes the advice associated with a matching cut. The first clause in the definition processes conditional advices using the function B, displayed in Table 11, to evaluate the condition. The second clause deals with non-conditional advices which are either **proceed** or **break** advices. In the former case, the before actions and after actions sandwich a recursive call to Φ to find further applicable aspects. In the latter case, the before actions are performed and Φ is called recursively to find further applicable aspects taking care to record the fact that a **break** has been encountered. Eventually, when all aspects in the aspect environment have been considered, the second clause of Φ is invoked (see Table 7). If no **break** has been encountered, the underlined action is emitted, otherwise a **stop** is emitted. In the latter case, the program will terminate after all of the before actions have been executed.

4 Example Programs

We now show a series of examples to illustrate how AspectK can be used to encode various security policies.

Example 4. The discretionary access control of Example 3 can be imposed by introducing a location DAC containing two kinds of triples

- $\langle user, \mathsf{DB}, \mathbf{read}\rangle$ for selected users, and
- $\langle user, name, \mathbf{out}\rangle$ for the same selected users and all names.

The following aspect declarations will then impose the desired requirements:

$$\begin{aligned}\mathsf{A}^{read}_{\mathsf{DAC}}[u :: \mathbf{read}(?x, ?y, ?z)@\mathsf{DB}] \triangleq\ & \mathbf{case}(\mathbf{test}(u, \mathsf{DB}, \mathbf{read})@\mathsf{DAC})\\ & \quad \mathbf{proceed};\\ & \quad \mathbf{break}\end{aligned}$$

$$\begin{aligned}\mathsf{A}^{out}_{\mathsf{DAC}}[u :: \mathbf{out}(z)@l] \triangleq\ & \mathbf{case}(\mathbf{test}(u, l, \mathbf{out})@\mathsf{DAC})\\ & \quad \mathbf{proceed};\\ & \quad \mathbf{break}\end{aligned}$$

The first action $\mathbf{read}(!name, !key)@\mathsf{YP}$ of User in Examples 1 and 2 will not be trapped by any of the aspects so it will simply be performed resulting in binding Alice to *name* and 55010 to *telno* as in Example 2.

The aspect $\mathsf{A}^{read}_{\mathsf{DAC}}$ will trap the second action in Examples 1 and 2 which now is

$$\mathbf{read}(55010, !val_1, !val_2)@\mathsf{DB}$$

The resulting substitution is $[\mathsf{User}/u, 55010/x, val_1/y, val_2/z]$ and we are evaluating the condition $\mathbf{test}(\mathsf{User}, \mathsf{DB}, \mathbf{read})@\mathsf{DAC}$. If this test evaluates to *false* then the advice **break** is taken and terminates the execution. Alternatively, we proceed and perform the action $\mathbf{read}(55010, !val_1, !val_2)@\mathsf{DB}$ thereby giving rise to the binding of 100 to val_1 and Bob to val_2.

Finally, the aspect $\mathsf{A}^{out}_{\mathsf{DAC}}$ will trap the last action which is now $\mathbf{out}(100)@\mathsf{Alice}$; also here the test will succeed and the **proceed** advice will be selected so that the original **out** is executed.

Using aspects it is easy to modify the access control policy so as to allow a user to access his own entries in DB even though he does not have access to the complete database. We simply modify the aspect $\mathsf{A}^{read}_{\mathsf{DAC}}$ to become

$$\begin{aligned}&\mathsf{A}^{read}_{\mathsf{DAC}-1}[u :: \mathbf{read}(!x, ?y, ?z)@\mathsf{DB}]\\ &\quad \triangleq \mathbf{break}\end{aligned}$$

$$\begin{aligned}&\mathsf{A}^{read}_{\mathsf{DAC}-2}[u :: \mathbf{read}(x, ?y, ?z)@\mathsf{DB}]\\ &\quad \triangleq \mathbf{case}(\mathbf{test}(u, \mathsf{DB}, \mathbf{read})@\mathsf{DAC} \vee \mathbf{test}(u, x)@\mathsf{YP})\\ &\qquad \mathbf{proceed};\\ &\qquad \mathbf{break}\end{aligned}$$

Example 5. To model the mandatory access control policy of Example 3 we introduce a location MAC with the following pairs:

- $\langle \mathsf{YP}, \mathsf{low} \rangle$ reflecting that the phonebook has low security level,
- $\langle \mathsf{DB}, \mathsf{high} \rangle$ reflecting that the customer database has high security level,
- $\langle s, \mathsf{low} \rangle$ for all users and names s with low security level, and
- $\langle s, \mathsf{high} \rangle$ for all users and names s with high security level.

We now consider the Bell-LaPadula security policy in a setting where both subjects and objects have fixed security levels. The first part of the policy states that a subject is allowed to read or input data from any object provided that the object's security level dominates that of the object; this is captured by the following aspects (which enforce *no read-up*):

$$\begin{aligned}\mathsf{A}^{read_2}_{\mathsf{MAC}}[u :: \mathbf{read}(?x, ?y)@l] \triangleq\ & \mathbf{case}(\neg(\mathbf{test}(u, \mathsf{low})@\mathsf{MAC} \wedge \mathbf{test}(l, \mathsf{high})@\mathsf{MAC})) \\ & \quad \mathbf{proceed}; \\ & \quad \mathbf{break} \\ \mathsf{A}^{read_3}_{\mathsf{MAC}}[u :: \mathbf{read}(?x, ?y, ?z)@l] \triangleq\ & \mathbf{case}(\neg(\mathbf{test}(u, \mathsf{low})@\mathsf{MAC} \wedge \mathbf{test}(l, \mathsf{high})@\mathsf{MAC})) \\ & \quad \mathbf{proceed}; \\ & \quad \mathbf{break}\end{aligned}$$

The second part of the policy, the star property, allows a subject to write to any object provided that the security level of the object dominates that of the subject. This is captured by the following aspect (enforcing *no write-down*):

$$\begin{aligned}\mathsf{A}^{out}_{\mathsf{MAC}}[u :: \mathbf{out}(z)@l] \triangleq\ & \mathbf{case}(\neg(\mathbf{test}(u, \mathsf{high})@\mathsf{MAC} \wedge \mathbf{test}(l, \mathsf{low})@\mathsf{MAC})) \\ & \quad \mathbf{proceed}; \\ & \quad \mathbf{break}\end{aligned}$$

With these aspects in place a user with low security level will only be able to perform the action $\mathbf{read}(!name, !key)@\mathsf{YP}$; once he attempts doing the action $\mathbf{read}(key, !val_1, !val_2)@\mathsf{DB}$ the advice **break** will stop the execution. A user with high security level will be able to perform both of these actions but may be stopped at the third action $\mathbf{out}(val)@name$ if the security level of the location bound to *name* turns out to be low.

In order to allow a high user to write to a low name we may introduce *declassification* of security levels. To keep things simple we may do so by introducing a billing location that does not need to adhere to the security policy and replace the process by:

$$\begin{aligned}\mathsf{User} :: \quad & \mathbf{read}(!name, !key)@\mathsf{YP}. \\ & \mathbf{read}(key, !val_1, !val_2)@\mathsf{DB}. \\ & \mathbf{out}(name, val_1, val_2)@\mathsf{Billing} \\ \| \ \mathsf{Billing} :: \ & \mathbf{in}(!n, !v_1, !v_2)@\mathsf{Billing}.\ \mathbf{out}(v_1)@n\end{aligned}$$

We add the pair $\langle \mathsf{Billing}, \mathsf{high} \rangle$ to the MAC location thereby allowing all high users to output to Billing; we also modify the aspect for **out** actions to ensure

that they are always allowed to **proceed** at the Billing location:

$$\mathsf{A}^{out}_{\mathsf{MAC}}[u :: \mathbf{out}(z)@l] \triangleq \mathbf{case}(\neg(\mathbf{test}(u, \mathsf{high})@\mathsf{MAC} \wedge \mathbf{test}(l, \mathsf{low})@\mathsf{MAC}) \vee (u = \mathsf{Billing})) \; \mathbf{proceed}; \; \mathbf{break}$$

Example 6. As a final example, which illustrates the need for actions both before and after **proceed** we define an aspect which maintains a log of **read** action on DB:

$$\mathsf{A}_{\mathsf{LOG}}[u :: \mathbf{read}(?x, ?y, ?z)@\mathsf{DB}] \triangleq \mathbf{in}(sem)@\mathsf{semaphore} \; \mathbf{proceed} \; \mathbf{out}(u, x, y, z)@\mathsf{logfile}. \; \mathbf{out}(sem)@\mathsf{semaphore}$$

We use a semaphore to ensure that the reads and the updating of the log file are kept in lock step, meaning that at any time at most one **read** action has been performed but still needs to be logged. The before action grabs the semaphore, **proceed** allows the read to be performed and the parameters that are bound in the read are recorded in the log file before the semaphore is released. In a similar way we can log **out** actions.

5 Conclusion

Summary. We have shown how to extend a coordination language with support for aspect oriented programming. While we have only performed the technical development for a fragment of KLAIM we do believe that our approach and our findings would apply to a larger class of coordination languages.

A distinguishing feature of coordination languages with respect to object oriented languages and web service languages [7] is the need to deal with *open* joinpoints, i.e. joinpoints that contain mechanisms for binding variables. Similar considerations would apply if we were to incorporate aspects into process algebras that, like the π-calculus, allow a notion of open input (or input from the environment) but would not be necessary for calculi without this feature [1, 6, 14, 22, 23]. This calls for considerable care in designing a notion of advice where input actions are trapped *before* a concrete tuple has been selected for input. We argued in the Introduction that the more standard choice of trapping an action after a concrete tuple has been selected would constitute a covert channel in the presence of open joinpoints. Our technical solution to this challenge was presented in Section 3 and we believe it to be applicable to open joinpoints in general.

In our development we focused on just two types of basic advice, **break** and **proceed**, together with actions performed before or after the advice (in order to obtain some of the benefits of **around** advice). We showed by means of examples that our approach is sufficiently flexible for defining aspects for enforcing discretionary and mandatory access control policies as well as mechanisms for logging actions. As argued in the Introduction we find this to be both a more flexible and less error-prone way of accomodating new security policies. Also we only considered the possibility of fixed global advice applicable at all locations.

There are different views as to whether the actions generated by an advice should also be subject to further advice. Throughout the development we have taken the view that this is indeed desirable. But it is straightforward to modify Table 10 to use underlined before and after actions so as to accommodate the alternative view.

Similarly, the use of a global test is often considered hard to implement because of the need to synchronise the whole network [18]. In our examples we have taken the view that we only perform tests on special persistant databases.

We now discuss the possibility of extending our design.

Types of advice. We did consider the incorporation of an **ignore** advice, as is commonly expressible in aspect oriented object oriented languages, but somewhat surprisingly found this to be a challenging extension.

To illustrate the problems consider the following advice

$$\mathsf{A}_{\mathsf{IGNORE}}[u :: \mathbf{read}(!v)@l_{priv}] \triangleq \mathbf{ignore}$$

for simply ignoring inputs from a private location l_{priv}. The problem with this definition is that it might be trapping a **read** action occurring in the following process $l :: \mathbf{read}(!w)@l_{priv}.\mathbf{out}(w)@l_{print}$ which would then become $l : \mathbf{out}(w)@l_{print}$ that contains a free variable; however, our semantics does not ascribe meaning to such processes!

Even a somewhat more useful advice

$$\mathsf{A}_{\mathsf{REDIRECT}}[u :: \mathbf{read}(!v)@l_{priv}] \triangleq \mathbf{ignore}\ u :: \mathbf{read}(!v)@l_{sandbox}$$

for redirecting inputs from a private location l_{priv} to a sandbox $l_{sandbox}$ is problematic. Once again consider the program $l :: \mathbf{read}(!w)@l_{priv}.\mathbf{out}(w)@l_{print}$ that is intended to become $l :: \mathbf{read}(!w)@l_{sandbox}.\mathbf{out}(w)@l_{print}$. The problem is that our current notion of substitution does not achieve this effect: while we can bind v to w to obtain the substitution $[w/v]$, we would not normally let the substitution change the defining occurrence $!v$ in $u :: \mathbf{read}(!v)@l_{sandbox}$ to $!w$ so as to yield the desired $u :: \mathbf{read}(!w)@l_{sandbox}$.

This can be solved by suitable extensions of our approach; in particular we can introduce special variables, e.g. β, that can be substituted also in defining

occurrences and write

$$\mathsf{A}_{\mathsf{REDIRECT}}[u :: \mathbf{read}(!\beta)@l_{priv}] \triangleq \mathbf{ignore}\ u :: \mathbf{read}(!\beta)@l_{sandbox}$$

Then the program $l :: \mathbf{read}(!w)@l_{priv}.\mathbf{out}(w)@l_{print}$ would correctly be transformed to $l :: \mathbf{read}(!w)@l_{sandbox}.\mathbf{out}(w)@l_{print}$.

Local or global advice. For simplicity we have taken an approach where all advice is given in advance and is global in scope. It would be worthwhile to be able to introduce new pieces of advice and to limit the scope of its applicability. Indeed, it might be natural to consider the aspect environment to be distributed and associated with locations. In that case it would be appropriate to extend the syntax with a $\mathbf{newloc}(u : \Gamma)$ construct with inference rule:

$$l ::^{\Gamma} \mathbf{newloc}(u : \Gamma').P \rightarrow l ::^{\Gamma} P[l'/u] \mid\mid l' ::^{\Gamma'} 0 \quad \text{with } l' \text{ fresh}$$

This would constitute a static treatment of scoped advice unlike the dynamic treatment in CaesarJ [2].

This would be useful when dealing with the **eval** action. Here we would extend the syntax of processes with a process identifier X that could match an arbitrary process. Then we might write an advice for executing a process in a sandbox as follows:

$$\begin{array}{l}\mathsf{A}_{\mathsf{SANDBOX}}[u ::^{\gamma} \mathbf{eval}(X)@l_{sensitive}] \triangleq \\ \quad \mathbf{newloc}(u_{sandbox} : \gamma[\mathsf{A}_{\mathsf{BOXREAD}}[u ::^{\gamma'} out(v)@w] \triangleq u ::^{\gamma'} \mathbf{out}(v)@u_{sandbox}]) \\ \quad \mathbf{ignore} \\ \quad u ::^{\gamma} \mathbf{eval}(X)@u_{sandbox}\end{array}$$

When executing a program $l ::^{\Gamma} \mathbf{eval}(P)@l_{sensitive}.P'$ the advice transforms it to a process that evaluates the process in a confined location and redirects all outputs to a confined location.

Clearly a number of additional extensions can be contemplated. For example we might want to have more powerful pointcut languages [4, 17] allowing patterns that bind over a number of parameters (in order to avoid having separate advice for each arity of the operations) or giving priorities to advice. However, our goal was to demonstrate both the need to, and the possibility of, dealing with open joinpoints.

Acknowledgements. Thanks to Sebastian Nanz for discussions about aspects. This project was partially funded by the Danish Strategic Research Council (project 2106-06-0028) "Aspects of Security for Citizens".

References

1. J. H. Andrews. Process-Algebraic Foundations of Aspect-Oriented Programming. *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference*, 2001.
2. I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.
3. P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2005.
4. P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 11–23, 2007.
5. L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim Project: Theory and Practice, 2003.
6. G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μabc: A minimal aspect calculus. In *Proceedings of the 2004 International Conference on Concurrency Theory*, pages 209–224. Springer, 2004.
7. Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In *ECOWS*, volume 3250 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2004.
8. D. S. Dantas. *Analyzing Security Advice in Functional Aspect-oriented Programming Languages*. Ph.D Thesis. Princeton University: Computer Science, 2007.
9. Ú. Erlingsson and F. B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.
10. S. Gao, Y. Deng, H. Yu, X. He, K. Beznosov, and K. Cooper. Applying Aspect-Orientation in Designing Security Systems: A Case Study. *The Sixteenth International Conference on Software Engineering and Knowledge Engineering*, 2004.
11. G. Georg, I. Ray, and R. France. Using aspects to design a secure system. *Engineering of Complex Computer Systems*, pages 117–126, 2002.
12. V. D. Gligor. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*. National Computer Security Center (US), 1994.
13. D. Gollmann. *Computer Security*. Wiley, 2006.
14. R. Jagadeesan, A. Jeffrey, and J. Riely. A Calculus of Untyped Aspect-Oriented Programs. *Ecoop Object-Oriented Programming: 17th European Conference*, 2003.
15. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Ecoop Object-Oriented Programming: 15th European Conference*, 2001.
16. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming*, pages 220–242, 1997.
17. H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. *Programming Languages and Systems: First Asian Symposium*, 2003.
18. R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *Transactions on Software Engineering*, 24(5):315–330, 1998.

19. R. De Nicola, G. Ferrari, and R. Pugliese. Programming access control: The KLAIM experience. In *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, pages 48–65. Springer-Verlag, 2000.
20. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
21. T. Verhanneman, F. Piessens, B. De Win, and W. Joosen. Uniform Application-level Access Control Enforcement of Organizationwide Policies. *Proc. 21st Annual Computer Security Applications Conference*, pages 431–440, 2005.
22. D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. *ACM SIGPLAN Notices*, 38(9):127–139, 2003.
23. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.
24. B. De Win, W. Joosen, and F. Piessens. Developing secure applications through aspect-oriented programming. *Aspect-Oriented Software Development*, pages 633–650, 2004.