From policies to aspects in KLAIM

Herbert, Luke Thomas; Egilsson, Einar

Published in:
Proceedings of The 13. Nordic Workshop on Secure IT Systems

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

From policies to aspects in KLAIM

Luke Herbert and Einar Egilsson

Department of Informatics, Technical University of Denmark
{lth,ee}@imm.dtu.dk

Abstract. The aspect oriented programming paradigm facilitates the separation of cross cutting concerns in system development. Security policies are a typical such concern and in this paper we present a simple policy language, LUNAR, and show how it can be translated into aspect definitions. We perform the development for KLAIM, a small kernel language for agent interaction and mobility, and show how static analysis can be used to limit the number of aspect definitions. The technique has been applied to a larger case study, namely the electronic invoice system at DTU.

1 Introduction

Motivation Modern business processes can be extremely complicated and the need exists to verify that these processes comply with a given security policy [H05]. This situation might arise in an invoicing system, where the processes put in place could allow embezzlement or other undesirable activity due to loopholes in the security policy, or in the business process itself. Today, the flow of many business processes is dictated by the software they employ, software which has often been required to quickly evolve as the needs of the business change, and which has not been built with security in mind.

The separation of concerns inherent in aspect oriented programming [MK03] allows security features to be developed separately from the main body of code, thus removing the complexity of adding such features and allowing them to change as needed. In addition, security features can be developed separately by people specialized in this field, hopefully ensuring a higher quality of code. However, the separation of code into aspects and the transformation of a security policy into aspects present many new challenges.

Contribution In Section 2 we introduce the subset of KLAIM to be used for specifying business processes. KLAIM is a kernel language for distributed computation where data and processes can be moved from one computing location to another [NFP98]. Processes can interact with one another by performing input, output and read operations and in Section 3 we present a simple policy

language called LUNAR that will be used to limit which operations can be performed at various stages in a workflow process. To enforce the policies we present in section 3.1, we make use of an aspect oriented version of KLAIM developed in [HNNY08]; in Section 4 we present a slight extension of the AspectK language together with an algorithm for transforming LUNAR policies into aspects; a subsequent weaving will then embed the aspects into the code itself. Finally, in Section 5 we present a static analysis that, given a KLAIM workflow process and a set of LUNAR policies will identify which policies might be violated; indeed it is only necessary to construct aspects for those. In Section 6 we give concluding remarks.

A overview of the basic design of LUNAR is shown in Figure 1.

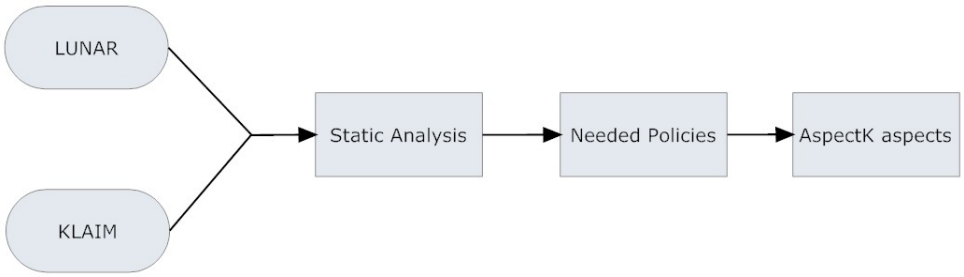

Fig. 1. The design of the LUNAR system

1.1 Case Study: the DTU invoice system

The above development has been carried out for a larger example modelling the workflow of the electronic invoice system at DTU; specifically how this system is used within the department of Informatics and Mathematical Modelling. Figure 2 shows the basic flow of the invoice process.

The typical flow of an invoice through the system is described below, this is the straightforward case of a purchase being approved with no problems arising:

1. A supplier sends a paper invoice to the scanning office which scans it to create an electronic invoice.
2. The scanning office forwards the electronic invoice to DTU's main finance department.
3. The finance department forwards the invoice to the section accountant of the relevant department.
4. In this business process the section accountant generally knows which person inside the department has been ordering which things, and so they forward the invoice to the person who placed the order.

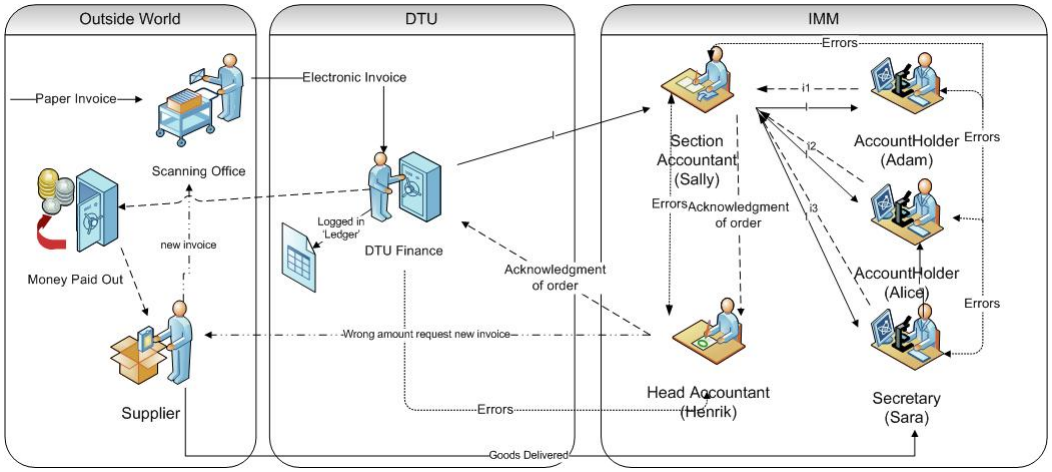

Fig. 2. Overview of how the DTU invoice system interacts with the IMM department

5. The person who ordered the product(s) confirms the invoice by adding their signature to it and then forwards the invoice to the relevant account holder.
6. The account holder approves the purchase by adding the correct account number to the invoice and sends it back to the section accountant.
7. The section accountant confirms that the invoice is correctly filled out, and then forwards it to the head accountant.
8. The head accountant confirms the invoice and forwards it to DTU Finance.
9. Finance pays the invoice and enters the amount into the main DTU ledger.

When errors or malicious activity occurs, other flows through the system take place. Some examples of alternative flows are described below:

- The section accountant sends an invoice to the wrong person. That person does not confirm the purchase but sends it back to the section accountant, who will then have to find out who the correct recipient is and forward it to him. In practice such errors accumulate to large amounts of wasted time.
- If the buyer sees that the price on the invoice is wrong, he can send a complaint to the supplier and wait for a credit note to be issued by the supplier, before sending both the invoice and credit note back to the section accountant. The invoice and credit note then both follow the normal path to get paid.
- If no one within the department recognizes an invoice payment, then it is refused and sent back to the supplier.

In this system a security policy can be enforced by controlling the flow of the invoice through the system. This is achieved by having each invoice carry a status

$N \in \mathbf{Net}$	$N ::= N_1 \,\|\|\, N_2 \;\mid\; l :: P \;\mid\; l :: \langle\, \overrightarrow{l} \,\rangle$
$P \in \mathbf{Proc}$	$P ::= P_1 \mid P_2 \;\mid \sum_i a_i.P_i \;\mid\; *P \mid$ if $cond$ then P_1 else P_2 fi
$cond \in \mathbf{BExp}$	$cond ::= \ell_1 = \ell_2 \;\mid\; test(\overrightarrow{\ell^\lambda})@\ell$
$a \in \mathbf{Act}$	$a ::= \mathbf{out}(\overrightarrow{\ell})@\ell \mid \mathbf{in}(\overrightarrow{\ell^\lambda})@\ell \mid \mathbf{read}(\overrightarrow{\ell^\lambda})@\ell$
$\ell, \ell^\lambda \in \mathbf{Loc}$	$\ell ::= u \;\mid\; l \qquad\qquad \ell^\lambda ::= \ell \mid !u$

Table 1. KLAIM Nets and Processes Syntax

tag which changes as it passes through the system. Each entity in the process should only receive invoices with the appropriate status. For example the scanning office should only receive *sent* invoices, DTU's finance department should only receive *presented* or *accounted* invoices and account holders should only receive *presented* or *confirmed* invoices. Other entities have similar restrictions which will be explained in detail in Section 3.1.

2 The KLAIM subset used in LUNAR

The LUNAR system analyses a subset of the KLAIM language [NFP98] similar to what is used in [HNNY08]. The syntax is defined in Table 1. A net N is a parallel composition of located processes or located tuples. Each person or entity in the business process is represented as a location. For a given entity, the collection of tuples present at its location is known as its tuple space, and the processes located at its location define its behaviour. A given location may have many processes running in parallel; each process can run just once or repeatedly, as indicated by the $*$ prefix. A process is made up of one or more actions and we allow just three operations, **in**, **out** and **read**. The **in** operation removes a tuple from a tuple space, **out** outputs a tuple to a tuple space and **read** reads a tuple from a tuple space but does not remove it. An operation parameter can be a location constant l, variable binding $!u$ or a variable u. A variable is defined when it is bound. All communication between entities in the business process is modelled by locations sending and receiving tuples from each other.

The **read** and **in** operations attempt to match a tuple to the tuple space they are reading from. The number of parameters in the operation must match the number of elements in a tuple for it to be selected. Furthermore, for any variable or constant parameter p_n in the operation, the element e_n in a tuple must have the same value for the tuple to be selected. This is used to conditionally select tuples from a tuple space.

In order to model conditional behaviour we have added a conditional expression to the language; in particular we needed to model different behaviours depending on the existence of a tuple at a particular location. The if statement can check for the existence of a tuple using the test function. The statement if $test(\overrightarrow{\ell^{\lambda}})@\ell$ then P_1 else P_2 fi will execute the P_1 process if there exists a tuple at ℓ that matches $\overrightarrow{\ell^{\lambda}}$, if the tuple does not exist process P_2 is executed instead.

A more precise definition of the language and its semantics, including its structural congruence and reaction semantics on closed nets can be found in [HNNY08].

2.1 The DTU invoice system expressed in KLAIM

For reasons of space we can not show the entire DTU invoice model here but include a representative example. The example shows the DTUFinance and Staff locations. Staff is a simple database mapping people to departments and roles, it has no processes. DTUFinance is the main DTU finance department which receives invoices and forwards them to other locations within the system.

```
DTUFinance :: <INVOICE, PRESENTED, 500, Fona, IMM, Printer, 200>
|| DTUFinance ::

   * in(INVOICE, PRESENTED, !nr, !supplier, !department, !item, !price)@self
   . read(!accountant, department, SectionAccountant)@Staff
   . out(INVOICE, PRESENTED, nr, supplier, department, item, price)@accountant
   |
   * in(INVOICE, ACCOUNTED, !invnr, !supplier, !department, !item, !price, !confirmedBy,
        !accountNr, !project)@self
   . out(INVOICE, PAID, invnr, supplier, department, item, price, confirmedBy, accountNr,
         project)@Ledger
   |
   * in(INVOICE, REFUSED, !invnr, !supplier, !department, !item, !price)@self
   . out(INVOICE, REFUSED, invnr, supplier, department, item, price)@supplier

|| Staff :: <Sally, IMM, SectionAccountant>
|| Staff :: <Adam, IMM, AccountHolder>
|| Staff :: <Alice, IMM, AccountSupervisor>
|| Staff :: <Sara, IMM, Secretary>
|| Staff :: <Henrik, IMM, HeadAccountant>
```

Code Example 1: DTUFinance location KLAIM code

Looking at DTUFinance in the code snippet above we see that it has one tuple located in its tuple space. The tuple format is $\langle$*type, status, invoice nr, vendor, department, item, price*$\rangle$. The DTUFinance location has three parallel processes. The first process removes an invoice tuple with the status `PRESENTED` from DTUFinance's tuple space, looks up the relevant department's section accountant by doing a **read** operation at the Staff location, and then outputs the tuple into the section accountant's tuple space. The second process pays invoices that have the status `ACCOUNTED` by removing them from DTUFinance's tuple space and out-

$SP \in$ **SecPolicy**	$SP ::= A\ G$
$A \in$ **Abbreviation**	$A ::= \$a = S \mid A\ ;\ A \mid \varepsilon$
$G \in$ **Group**	$G ::= "name"\ P \mid G\ ;\ G$
$P \in$ **Pattern**	$P ::= P\ ;\ P \mid V{::}\mathbf{out}(\overrightarrow{V})@V \mid V{::}\mathbf{in}(\overrightarrow{V})@V \mid V{::}\mathbf{read}(\overrightarrow{V})@V$
$V \in$ **Value**	$V ::= S \mid [S]$
$S \in$ **Set**	$S ::= S + S \mid t$
$t \in$ **Terminal**	$t ::= l \mid \$\mathrm{a} \mid\ *\ \mid\ ...$

Table 2. LUNAR Syntax

putting them to Ledger with status `PAID`. The third process removes `REFUSED` invoice tuples and sends them back to the supplier that they originated from.

In this case the first process finds a match since it looks for invoices with status `PRESENTED` and then forwards the invoice to the relevant location. Neither the second nor third process find any tuples matching their **in** operations in DTUFinance's tuple space. Other entities within our system are modelled in a similar way.

3 The LUNAR policy language

LUNAR allows the user to express security policies using the LUNAR policy language, which is simple formal policy language inspired by regular expressions [LP97]. The syntax for the language is shown in Table 2. A given security policy is made up of zero or more *abbreviations* and one or more *pattern groups*.

Abbreviations allow a large set of values to be represented concisely and can be used in more than one place in a security policy. They are defined before any patterns and cannot be redefined later in the policy. Abbreviations start with a $ character to distinguish them from ordinary values. An example of an abbreviation is:

```
$AccountHolders=Adam+Alice
```

Pattern groups are defined after the abbreviations. Each pattern group is comprised of a name and one or more patterns. The patterns themselves are similar to KLAIM statements, each pattern is made up of a sender, an operation, one or more parameters, and a receiver.

$sender :: op(p_1, p_2, ...p_n)@receiver$

Each component of the pattern (*sender*, *op*, p_n, *receiver*) can be expressed as a single value or as a set of possible values for that component. A set is a list of the possible values separated by +, so for instance the *sender* part of the pattern could have the value Adam+Henrik+Sara. Pattern components are matched against action parameters to see if the pattern should apply to that action, except in the case where a component is surrounded by [ ], in that case the component represents allowed values for the actions' parameter. The * token is a wildcard character matching any possible value, and the ... token can be thought of as one or more *'s, and can only be the last parameter in a parameter list. This is introduced to handle the case where tuples gain additional elements in the course of the workflow, yet we still require that the pattern matches regardless of the number of elements.

3.1 Case study: DTU security policy

The purpose of our security policy is to prevent information leaks, that is to say invoices reaching unintended recipients. Such a security policy can be enforced by controlling the flow of invoices through the system, by use of the status tag. Specifically we create LUNAR patterns which specify the states invoices and credit notes may be in when they are output into a particular locations tuple space. The LUNAR policy language allows for role based access control [G99], to implement this we start by creating abbreviations that define the roles within the system.

```
$Vendors=HP+OfficeSupplies+TaxiStation
$SectionAccountant=Sally
$Secretary=Sara
$AccountHolders=Adam+Alice
$HeadAccountant=Henrik
```

Using these roles, expressed as abbreviations in LUNAR policy notation, we can define the following security policy:

1. **Vendor allowed content:** Vendors can only receive complaints, invoices and creditnotes. Specifically the invoices may only have the status *sent* or *refused* and creditnotes may only have the status *sent*.
2. **ScanOffice allowed content:** The scanning office can only receive invoices and creditnotes and they may only have the status *sent*.
3. **DTUFinance allowed content:** DTU's finance department can only invoices and creditnotes. Specifically the invoices may only have the status *presented*, *accounted* or *refused* and creditnotes may only have the status *presented* or *accounted*.
4. **Ledger allowed content:** Only DTU's finance department may output tuples into the ledger. The tuples may be either invoices and creditnotes and they may only have the status *paid*.

5. **SectionAccountant allowed content:** Section accountants may only receive invoices and creditnotes and they may only have the status *accounted* or *presented*.
6. **Secretary allowed content:** The secretary can only receive complaints and invoices. Specifically the invoices may only have the status *presented* or *wrong* and creditnotes may only have the status *presented*.
7. **AccountHolder allowed content:** Account holders can only receive complaints and invoices. Specifically the invoices may only have the status *presented*, *confirmed* or *wrong* and creditnotes may only have the status *presented* or *confirmed*.
8. **Head Accountant allowed content:** The head accountant can only receive complaints and invoices with the status *accounted*.

The above rules are each mapped into a pattern group consisting of one or more patterns. Generally the pattern groups start with a line defining what types of tuples are allowed and subsequent lines restrict what states each type of tuple may be in when it is output into the entity's tuplespace.

Next we present two examples of rules expressed as LUNAR pattern groups. Rule 4 only allows DTUFinance to output paid invoices and creditnotes into the Ledger. This stops employees from trying to slip paid invoices past the finance department. This can be expressed as a single pattern since the invoice and creditnote have the same allowed states.

```
"4. Ledger allowed content"
[DTUFinance]::out([INVOICE+CREDITNOTE], [PAID], ...)@Ledger
```

A more complex pattern group is required to implement rule 7. In this case the invoice and creditnote have different allowed states, the first pattern restricts the type of tuples that can be sent to account holders, the second and third pattern restrict the allowed states for invoices and creditnotes respectively.

```
"7. AccountHolder allowed content"
*::out([INVOICE+CREDITNOTE], ...)@$AccountHolders
*::out(INVOICE, [PRESENTED+CONFIRMED+WRONG], ...)@$AccountHolders
*::out(CREDITNOTE, [PRESENTED+CONFIRMED], ...)@$AccountHolders
```

4 Conversion of policies to aspects

An aspect oriented coordination language, AspectK, was introduced in [HNNY08]. This language is a superset of the KLAIM like language we have used to model the DTU invoice system, and supports the addition of aspects. The AspectK language takes the approach that input actions should be trapped before a concrete tuple has been selected for input, this is very useful in the LUNAR system where

$S \in \mathbf{System}$	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	$::= \ell \mid !u \mid ?u \mid \ldots$

Table 3. AspectK syntax

we are focused on checking a security policy. If we were to trap after a concrete tuple has been selected for input, it would constitute a covert channel [G99] as the presence or absence of a tuple in the tuple space might enable or prevent the advice to trap the action, and this would amount to visible behaviour bypassing the security policy. In [HNNY08] a successful method for trapping an input action before a concrete tuple has been selected for input which has an ability to deal with joinpoints that contain constructs for binding new variables. Such joinpoints are referred to as 'open joinpoints' in [HNNY08] and we will do the same. AspectK aspects allow a number of actions for handling a trap including the option to to block an action or to let it proceed which is sufficient for the LUNAR system. For these reasons we have chosen to use a slightly modified version of AspectK, into which to transform our security policy, the modifications we have introduced are very small and will be introduced in the description of the AspectK syntax below.

4.1 AspectK

The AspectK syntax is an extension of the subset of the KLAIM syntax shown in Table 1. AspectK aspects have two main elements, the *cut* and the *body*. The *cut* is the part that is matched against actions being executed and decides whether the aspect in question should be applied to the executing action. If the *cut* matches the action then the *body* of the aspect is executed. The *body* has to contain either a break statement which stops the action from being executed, or a proceed statement which allows the execution to proceed. The aspect body can also have actions that are executed before a break statement, or before and after a proceed statement. Variables that are bound in the action can only be accessed in the body after a proceed statement, since before the break or proceed statements a concrete tuple has not yet been selected, and hence the variables

have not been bound. AspectK also adds a new token for ℓ^{λ}, the $?u$ token which matches both l and $!x$ in actions.

The trapping of actions and execution of aspects is done by the functions shown in tables 4 through 7. For a more detailed discussion of how these functions work we refer to [HNNY08], we show them here with our modifications for the sake of completeness, and to discuss what modifications we have made to allow wildcards and error messages to be incorporated into AspectK.

The ϕ function in Table 4 checks each action against every aspect. It uses the *trap* function from Table 5 to check whether the aspect matches the action. The *trap* function then uses the *check* function from Table 6 to determine whether the entities from the cut match the entities in the action, and produce a substitution of values for variables. If the aspect matches the action then the aspect body is executed before checking the next aspect. After checking every aspect it looks at the index f to determine whether to execute the original action or halt. If any aspect has issued the break statement, then the action is not executed and the process halts.

$$
\begin{array}{rl}
\Phi_f(A[cut] \triangleq body, \Gamma_A; \ell :: a) = & \text{case } trap(cut, \ell :: a) \text{ of } \mathsf{fail}: \quad \Phi_f(\Gamma_A; \ell :: a) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \theta: \quad \kappa_f^{\Gamma_A, \ell::a}(body\ \theta) \\
\Phi_f(\varepsilon; \ell :: a) = & \text{if } size(f) > 0 \text{ then: } \mathbf{stop}(f) \\
& \qquad\qquad\qquad\quad \text{else: } \underline{a}
\end{array}
$$

Table 4. Trapping Aspects: Step 1.

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

Table 5. Trapping Aspects: Step 2.

We have made two modifications to the original AspectK definition of these functions. First, we modified the *check* function to accommodate the ... wildcard character. Our version deviates from the original in that the number of parameters does not have to be the same in both tuples, there can be k entities in the cut but $k + n$ entities are possible in the action and happens if the ... wildcard is used. The function checks the first entity in the cut tuple against

$$check(\langle\rangle, \langle\rangle) = id$$

$$check(\langle \ldots \rangle, \langle \ell_1'^{\lambda}, \cdots, \ell_n'^{\lambda} \rangle) = id$$

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

Table 6. Trapping Aspects: Step 3.

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

$$\begin{aligned} \kappa_f^{\Gamma_A, \ell::a}(sbody) = \text{case } sbody \text{ of } \quad & as_1 \ \textbf{proceed} \ as_2 : as_1.\Phi_f(\Gamma_A; \ell :: a).as_2 \\ & as \ \textbf{break(msg)} \ : as.\Phi_{f \cup \{\textbf{msg}\}}(\Gamma_A; \ell :: a) \end{aligned}$$

Table 7. Trapping Aspects: Step 4.

the first entity in the action tuple, and produces a substitution if they match or fails if they don't. It then recursively calls itself using the tuples with the first entity removed. If both tuples are empty the function returns indicating that the aspect did match the action. If the cut tuple has only the ... wildcard and the action tuple has n entities left, the function also returns indicating that the aspect is a match, regardless of the values of the n entities left in the action tuple.

Secondly, we changed the index f and the break statement. In the original version f could take on either the value break or proceed. In our version we allow the break statement to take a parameter, which is an error message saying why the action is not permitted. We then define f to be a set of error messages, and in ϕ we check whether the number of elements in the set f is greater than zero, if so we halt the process and display the error messages in f to the user.

4.2 Converting LUNAR patterns to AspectK aspects

The LUNAR system can read a security policy specified in the LUNAR policy language and convert the patterns to AspectK aspects. Each generated aspect consists of two conditional statements. The outer conditional checks whether the matching values (values not inside [ ]) in the pattern match their corresponding

elements in the action. If they do not match then the pattern is not a match for the action, and a **proceed** advice is given. If the pattern matches the action, then the inner conditional statement will check for each of the allowed values parameters whether the action contains allowed values. If the action contains any value that is not in the set of allowed values, then the violation is logged to a Log location and a **break** advice is given. The * wildcard in a pattern is transformed to a $?p_n$ parameter in the AspectK cut, so it can match both location constants and binding of variables, as the p_n parameter can contain anything it is not checked. We have also added the ... token to AspectK cuts, where it has the same meaning as in the LUNAR policy language, any number of parameters with any values.

We employ the following algorithm to convert LUNAR policies to Aspects:

The Algorithm
1) Given a pattern u::op(p_1,p_2, ..., p_n)@l
2) Create tuple t as $\langle u, p_1, p_2, ... p_n, l\rangle$
3) Create outer case condition $case_{outer}$
4) Create inner case condition $case_{inner}$
5) for each part p_i in t:
  if (p_i is V_1+V_2+...+V_k)
    create new or-clause $match_i$
    for each value v_j in p
      add condition '$p_i = v_j$' to $match_i$
    add $match_i$ to $case_{outer}$ condition
    add parameter p_i to aspect cut
  else if (p_i is [V_1+V_2+...+V_k])
    create new or-clause $allowed_i$
    for each value v_j in p_i
      add condition '$p_i = v_j$' to $allowed_i$
    add $allowed_i$ to $case_{inner}$
    add parameter p_i to aspect cut
  else if (p_i is *)
    add parameter $?p_i$ to aspect cut
  else if (p_i is ...)
    add parameter ... to aspect cut

4.3 Case study: transformation example

The transformation shown below is of the pattern that specifies which tuples can be put into Ledger's tuplespace. Ledger represents DTU's general ledger and should only receive invoices and credit notes which are paid, and only from DTUFinance.

```
"4. Ledger allowed content"
[DTUFinance]::out([INVOICE+CREDITNOTE], [PAID], ...)@Ledger
```

is transformed into

$$
\begin{aligned}
\mathsf{A}[user :: \mathbf{out}(p1, p2, ...)@loc] \triangleq\ & \mathbf{case}(loc = Ledger) \\
& \quad \mathbf{case}(p1 = INVOICE \vee p1 = CREDITNOTE) \\
& \quad \wedge (p2 = PAID) \wedge (user = DTUFinance) \\
& \quad\quad \mathbf{proceed} \\
& \quad ; \\
& \quad\quad \mathbf{out}(Violation, user, p1, p2, loc)@Log \\
& \quad\quad \mathbf{break}\ \text{"Ledger allowed content"} \\
& ; \\
& \quad \mathbf{proceed}
\end{aligned}
$$

For **out** actions this transformation is enough and results in one aspect for each pattern. For **in** and **read** actions however we also have to consider that an action might try to bind a variable in a position where we have specified allowed values. Consider the case where we have the pattern

```
"C allowed reads"
*::read([A], [B], *)@C
```

Here we state that a **read** operation can only be performed at location C if the first parameter has the value A and the second parameter has the value B. The parameter values in the actual action can be checked against the allowed values if they are location constants or variables, but if they are variable bindings then they they will not match the AspectK cut. Consider the action

```
User::read(!x, !y, !z)@C
```

This will not match the generated AspectK cut A[u::**read**(p1, p2, ?p3)@l], and would allow the user to randomly bind to a tuple in C's tuplespace. In the case where allowed values are specified for a parameter we want to break if a user tries to bind a variable to that parameter. To do that we need the pattern in question to issue a **break** advice on all of the following actions:

```
U::read(AA, BB, !z)@C
U::read(!x, B, !z)@C
U::read(A, !y, !z)@C
U::read(!x, !y, !z)@C
```

The first action can be checked with an aspect generated from the transformation described above. For the three remaining aspects which all contain variable bindings, we generate one aspect for each possibility. These aspects only check whether the matching values in the pattern (in this case only C) match the action and then immediately break. The three extra aspects generated in this case would be

$$\mathsf{A}[user :: \mathbf{out}(!p1, p2, ?p3)@loc] \triangleq \mathbf{case}(loc = C) \; \mathbf{out}(Violation, u, p2, l)@Log \; \mathbf{break} \text{ "C Allowed reads"} \; ; \; \mathbf{proceed}$$

$$\mathsf{A}[user :: \mathbf{out}(p1, !p2, ?p3)@loc] \triangleq \mathbf{case}(loc = C) \; \mathbf{out}(Violation, user, p1, loc)@Log \; \mathbf{break} \text{ "C Allowed reads"} \; ; \; \mathbf{proceed}$$

$$\mathsf{A}[user :: \mathbf{out}(!p1, !p2, ?p3)@loc] \triangleq \mathbf{case}(loc = C) \; \mathbf{out}(Violation, user, loc)@Log \; \mathbf{break} \text{ "C Allowed reads"} \; ; \; \mathbf{proceed}$$

It is possible in AspectK to define a cut which matches all these possibilities, by having all parameters take the form $?p$, since parameters prefixed with ? match both location constants and variable bindings. So a cut such as A[u::**read**(?p1, ?p2, ?p3)@l] would indeed match any possible combination of variable bindings and location constants. The problem with this approach is that in the body of the aspect we have no way of determining whether $?p1$ and $?p1$ are variable binding or not. Because of that we cannot check their values since if they are in fact variable bindings they will have no values and the variables $p1$ and $p2$ should not be used in the aspect body until after a **proceed** statement.

5 Static Analysis

We employ static analysis, inspired by [NGHNNPP08] and [RPN08], to determine a reduced set of aspects needed to enforce a given security policy. The static analysis we perform works by executing the following algorithm on a KLAIM code file and an associated LUNAR policy file.

We define the following global variables for the algorithm; T_l the set of all tuples that can reside in the tuple space of a given location l. For each location's (l) processes' (p) we define σ_{lp} to denote the set of values that each variable in the process can take. Further we define the function *comb*(σ, $\overrightarrow{\ell}$) to return all possible variations of $\overrightarrow{\ell}$ by using all possible combinations of variable substitutions from σ.

Static Analysis Algorithm
1) For each location l
   Initialise T_l with the existing tuples at l.
2) Define $T' := T$ and $\sigma' := \sigma$.
3) For each locations' l processes' p action a

if $a = out(\overrightarrow{\ell})@l_{out}$
$T_{l_{out}} := T_{l_{out}} + comb(\sigma_{lp}, \overrightarrow{\ell})$
if $a = in/read(\overrightarrow{\ell})@l_{in}$
Get possible matches from $T_{l_{in}}$
Update σ_{lp} for all $!x$ in $\overrightarrow{\ell}$
4) If $T' \neq T$ or $\sigma' \neq \sigma$ goto 3.

This algorithm works by letting the tuples at each location propagate to all the locations that they can possibly reach limited by the actions at each location. The algorithm halts as the possible values of all tuples in KLAIM are finite sets and as such the sets T and σ are finite, and eventually the stabilisation condition in step 4 will be satisfied.

To apply the analysis we define K_{lpa} which denotes the possible values for a given locations' processes' actions' variables as specified by LUNAR policies. Further we define the function *match*(α, K_{lpa}) which returns true if the lunar pattern α matches K_{lpa}. Finally, we define A to be the set of patterns that need to be implemented as aspects. We then employ the following algorithm:

Applying the analysis
1) For each locations' l processes p action a
Use σ_{lp} to generate all possible variations of a and add them to the set K_{lpa}
2) For each lunar pattern α
For each location's l processes p action a
if $match(\alpha, K_{lpa})$ then
$A := A + \alpha$

This algorithm compares the possible parameter values that can occur with the ones that are permissible given a specific security policy. The test in step 2 is the key part and allows the set A to be filled with policies that may be violated.

6 Conclusion

We have presented the LUNAR system, and the LUNAR policy language for specifying security policies for KLAIM nets. This system has been implemented, tested and source code has been made available. This is a full system for adding security policies to KLAIM nets and serves as a proof of concept of a method for determining the elements of a security policy that are necessary to implement as aspects.

Given the power of aspect oriented programming, if a programmer makes a logical mistake when writing an aspect it can lead to widespread program failure. Therefore, reducing the set of aspects that need to be implemented should help shrink the chance of one containing a mistake. The LUNAR system serves as a

proof of concept that this can indeed be done and has been presented in general enough terms to be applied to other languages.

While the LUNAR system operates on KLAIM code, in future work we would like to extend the system to allow the application of a reduced set of aspects to other code, such as Java code with aspects implemented as AspectJ aspects [KLMMLLI97] and analysis done as in [ACHKLLMSST05].

A Source code

This report, source code for the LUNAR system and pre-compiled Windows binaries are available from http://www.student.dtu.dk/~lthhe/.

References

[H05] Michael Havey. Essential Business Process Modeling. *O'Reilly Media; 1st edition*, 2005.

[MK03] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. *Programming Languages and Systems: First Asian Symposium*, 2003.

[ACHKLLMSST05] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. *ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM Press*, 2005.

[KLMMLLI97] 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 220242*, 1997.

[HNNY08] Chris Hankin, Flemming Nielson, Hanne Riis Nielson and Fan Yang, Advice for Coordination. *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*

[RPN08] Ren Rydhof Hansen, Christian W. Probst and Flemming Nielson Sandboxing in myKlaim. *Proceedings of the The First International Conference on Availability, Reliability and Security, ARES 2006, The International Dependability Conference - Bridging Theory and Practice, April 20-22 2006, Vienna University of Technology, Austria*

[NGHNNPP08] Rocco De Nicola, Daniele Gorla, Ren Rydhof Hansen, Flemming Nielson, Hanne Riis Nielson, Christian W. Probst, and Rosario Pugliese. From Flow Logic to Static Type Systems for Coordination Languages. *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, 2008.

[NFP98] R. De Nicola, G. Ferrari and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *Transactions on Software Engineering, 24(5):315330*, 1998.

[G99] Dieter Gollmann. Computer Security. *John Wiley & Sons; 1st edition*, 1999.

[LP97] H. Lewis and Christos Papadimitriou. Elements of the Theory of Computation (2nd ed). *Prentice-Hall*, 1997.