Expressing access control policies with an event-based approach

Pierre Konopacki, Marc Frappier, Régine Laleau

Expressing access control policies with an event-based approach

Pierre Konopacki **Marc Frappier** **Régine Laleau**

March 2010

TR–LACL–2010–6

Laboratoire d'Algorithmique, Complexité et Logique (LACL)
Département d'Informatique
Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle, 94010 Créteil cedex France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01

Laboratory of Algorithmics, Complexity and Logic (LACL)
University Paris 12 (Paris East)

Technical Report **TR–LACL–2010–6**

P. Konopacki, M. Frappier, R. Laleau.
Expressing access control policies with an event-based approach

Expressing access control policies with an event-based approach

Pierre Konopacki **Marc Frappier** **Régine Laleau**

Laboratory of Algorithmics, Complexity and Logic - University Paris 12 (Paris East), France
Groupe de Recherche en Ingéniérie du Logiciel - Sherbrooke University, Canada

{pierre.konopacki,laleau}@univ-paris12.fr
{pierre.konopacki,marc.frappier}@usherbrooke.ca

Abstract

In this paper, we introduce EB^{3}SEC. This language is used to express access control policies in information systems. Permissions and prohibitions are expressed with a class diagram. EB^{3}SEC also includes a process algebra. This process algebra allows one to express specific constraints over permissions and prohibitions. Organizational constraints such as obligation and separation of duty are also supported by process algebra. Separation of duty constraints can be expressed at a workflow process level. Standards such as RBAC or OrBAC can be used to express the access control policy, but their derivatives can also be used. EB^{3}SEC provide a formal language with a high level of expressiveness to describe access control policies. **Keywords.** Access Control, Formal Security Methods, Security Models

1 Introduction

Information Systems (IS) are widely used in various economical and social areas. They contain private and valuable data for their owners. In an IS security is enforced by many mechanisms, such as secured protocols used between clients and servers or between servers in a distributed architecture. The most used architecture to develop IS is the Service Oriented Architecture (SOA). Many mechanisms have also been developed to secure different components of an IS build with this architecture, or to secure communication between those components [14]. All those tools deserve one purpose : to enforce an access control (AC) policy. The AC policy is the part of the security policy that deals with authorizations granted to users. An AC policy badly defined can lead to major issues for the company that uses the IS : as an example we can cite banks that lost billiards since a trader used more authorization that he should have been granted.

Security is an important goal for IS designers. In IS, data integrity is insured by an *access control* (AC) policy which is implemented in the software and enforced by mechanisms like encryption, secure data transfer protocols and authentication protocols. Furthermore, in some domains, IS security is regulated by laws. As an example, in the financial domain, we can cite the Sarbane-Oxley law in the United-States [23] and the Mer law in France [19]. In health care area, we can cite the PIPEDA law in Canada [13]. These laws aim to protect information by regulating their access. Organisations have to prove that the AC policy used by their IS comply with these laws. In this paper we propose a new formal method which aims to model AC policies.

Our project is conducted with an industrial partner from the banking industry in Canada. The management of security policies in an industrial context is a highly complex task. Security policies

are informally described using plain English and adhoc diagrams, and then they are implemented into concrete programs, in various components. Generally, the AC policy is implemented directly in the individual services, mixed with the business logic of the IS. The traceability between the natural language specification and the implementation code is hard to maintain. Thus, modifying and validating an AC policy is a hard task. A small change in AC policy may easily take up to a month of effort for implementation. With the current turmoil about security in the industry, security policies are bound to be frequently changed, which induces high maintenance costs for organisations.

Our goal is to streamline the management of AC policies. In this paper, we define EB^{3}SEC: a simple, abstract, formal specification language to describe AC policies. To describe the use of EB^{3}SEC we give some example of how common AC constraints can be expressed in EB^{3}SEC. The idea is to develop security filter based on EB^{3}SEC modeling. The first step to achieve this this goal is to show a prototype of tool using EB^{3}SEC. We choose to present an interpretor that uses an EB^{3}SEC specification to grant or refuse incoming events.

The paper is structured as follows. Section 2 explains the problem addressed by EB^{3}SEC. Section 3 presents the EB^{3}SEC language and the constraints that can be expressed in an EB^{3}SEC model. The expressiveness of the language is illustrated with an example presented in 3.1. After a discussing about choices during the modeling in 4, we present a possible implementation of an EB^{3}SEC interpretor in 5. In 6, we present future works related to EB^{3}SEC, before comparing EB^{3}SEC to other similar tools in 7.

2 PROBLEM DEFINITION - PROBDEF

We aim to create a language to model AC policies. A lot of methods with the same purpose already exist. These methods have a state based approach [6]. In other words, expressing an AC constraint only implies the current state of the system. Other methods, which are not state based, focuses on a particular kind of constraint [18, 24]. Case studies depicting AC policies used by our industrial partners imply business processes and constraints over actions ordering. Due to the variety of constraints that can be found in the case studies, the language we have to create must have a high level of expressiveness.

In order to ease the modeling of these case studies, we propose a different approach based on events accepted by the system, but with keeping a formal aspect. To do so, we want to express an AC policy, by describing constraints it involves on events accepted by the system.

3 MODELING ACCESS CONTROL POLICIES - MODACCCONTPOL

In this section we present some notations and assumptions used in EB^{3}SEC. The denotational semantics of an EB^{3}SEC specification is given by a relation R defined on $\mathcal{T}(\texttt{main}) \times O$, where $\mathcal{T}(\texttt{main})$ denotes the traces accepted by `main` and O is the set of output events. The operational behaviour of `main` is defined as follows. Let `trace` denote the system trace, which is a list comprised of *valid* input events accepted so far in the execution of the system. Let $t::\sigma$ denote the right append of an input event σ to trace t, and let `[ ]` denote the empty trace.

`trace` := `[ ]`;
forever do
    receive input event σ;
    if `main` can accept `trace::`σ **then**

$\texttt{trace} := \texttt{trace}::\sigma$;
send output event o such that $(\texttt{trace}, o) \in R$;
else
send error message;

We assume that to execute an action in the system, a user must be logged in. Thus, for each events, we know the person trying to execute it, the role this person played, the place and the time it happened. We introduce the notion of security events, which are n-tuples. The last element of the n-tuple is the event itself, the other elements contains values of some parameters. For this paper, we choose to use 5-tuples :

$$\sigma = \langle p, r, o, t, evt \rangle$$

. The definition of the 5-tuple is given by :

$\langle p, r, o, t, evt \rangle$with:
$p \in personId$
and $r \in roleId$
and $o \in orgId$
and $t \in Timestamp$
and $label(evt) \in action$

In this definition $User.name()$ is the set containing all values available for users during the login step ; by the same way we describe $Role.name()$, $Branch.name()$ and $Action.label()$ which respectively contain all the values of roles, organizations and actions. In our definition, we consider that the time is represented by an integer which means the timestamp of the moment the action is performed. As *evt* is an event we use the function *label* which return the name of the action instantiated. *evt*, the last part of a security event, is an instantiation of (the input parameters of) an action. The signature of an action a is given by a declaration

$$\mathsf{a}(q_1 : T_1, \ldots, q_n : T_n) : (q_{n+1} : T_{n+1}, \ldots, q_m : T_m)$$

where $q_1, \ldots, q_n$ are input parameters of types $T_1, \ldots, T_n$ and $q_{n+1}, \ldots, q_m$ are output parameters of types $T_{n+1}, \ldots, T_m$. A security event $\langle p, r, o, t, \mathsf{a}(t_1, ..., t_n) \rangle$ also constitutes an elementary process expression. The special symbol "_" may be used as an actual parameter of an action, to denote an arbitrary value of the corresponding type.

Complex EB^{3}SEC process expressions can be constructed from elementary process expressions (instantiated actions) using the following operators: sequence (**.**), choice ($|$), Kleene closure (*), interleaving ($|||$), parallel composition ($\|$, *i.e.,* CSP's synchronisation on shared actions), guard ($\Longrightarrow$), process call, and quantification of choice ($|x \in T : \ldots$) and interleaving ($|||X \in T : \ldots$). The EB^{3}SEC notation is similar to CSP [15] but the main differences between EB^{3}SEC and CSP are: i) EB^{3}SEC allows one to use a single state variable, the system trace, in predicates of guard statements (as we shall see below); ii) EB^{3}SEC uses a single operator, concatenation (as in regular expressions), instead of prefixing and sequential composition, which makes specifications easier to read and write.

In the following we use this process algebra to express AC policies. We give now the kind of constraints that can be expressed in an AC policy modeled in EB^{3}SEC. Those constraints will be more precisely described in the following of the document.

Permission allows the execution of an action.

Prohibition forbids the execution of an action.

Obligation forces the user to execute an action after an event

Separation of duty are used to forbid an execution of an action for a user after an event.

Now, we discuss about each of these kinds of constraints, after introducing a running example to illustrate our discussion.

3.1 THE RUNNING EXAMPLE : THE CHECK DEPOSIT - RUNEX

Our industrial partner provides us some case studies to work on. These case studies deal with check deposit in a bank branch. Thus, as in [4] and [22] we illustrate our work on check deposit example.

We consider here the following process of the check deposit : Assuming that the client already has an account in the company, he brings a check to the branch where he is affiliated. A bank employee makes the check deposit effective in the IS. Thus, this deposit could be canceled, or validate, in the second case, the account of the client has to to be credited.

The main difference with [4] and [22] was to consider an access control policy over different bank branches. As an example, permissions given to users are different from each other depending on where the branch is located. Considering a procedure of check deposit, generally it has to be validated by a supervisor, the number of required validations depends on the amount of the check deposit, and this value fluctuates depending on the state where the procedure takes place. In order to be more precise and understandable we stipulate in the following access control rules taking place in the example.

rule 1 : Only clerk and banker are allowed to make check deposit effective.

rule 2 : Only banker and chief agencies are allowed to cancel or validate a check deposit.

rule 3 : Only clerk and banker are allowed to modify the amount of a bank account.

rule 4 : The validation or the cancellation of a deposit can not be done by the same person that did the check deposit.

rule 5 : If the value of a check deposit exceed an amount, the check deposit must be validated by two different persons including the chief agency. In Quebec this amount equals 10 000 $ whereas in Ontario this amount equals 8 000 $.

rule 6 : The modification of the bank account of the customer must be done by the same employee that did the check deposit effective.

In this example, we use four different actions which are deposit, cancel, validate credit. Those actions have three arguments which refer to the customer, the number of the check and the value of the check deposit. In our model, we use integer for the value of the check, this assumption could be easily understand, as a value of a check has only two digits after the coma. In this example, people can play four different roles : *customer*, *clerk*, *banker* and *chief agency*. We also assume that we have at least two bank branches one in Montreal, QC and the other in Toronto, ON.

3.2 PERMISSION

When giving a permission, we want to allow someone to do something, but we want the security environment to comply with a specific pattern. Thus, dealing with permission, means to establish pattern for the security environment for each event. By pattern, we mean that in order to be executed, an event must have a security environment that respect some specific value.

The EB^{3}SEC language allows the modeler to express the AC policy he wishes. This policy can comply with an existing AC model as RBAC which is an ANSI standard [8]. The modeler can also use a hand made model. To model the running example, we use the second method. In fact, the example centralizes in a unique model AC policies of different branches. However, AC policy used in each branch is quietly similar to the others, it slightly differ. To achieve this task we add to the RBAC standard, the notion of organization which was first created in the OrBAC model [**?**]. Generally by organization, people think of AC constraints involved in SoD problems. In the OrBAC model, it deals with the location. As an example, it could be used to model different branches of the system. In order to be more readable, we use the term *Branch* in our model in place of *organization*. In comparison to the RBAC standard, this notion of *branch*/*organization* avoid us to multiply the number of instances of actions and role. For example if we did not used this concept we had to create two different roles for customer : one for each branch.

In order to express permissions, we have the 5-tuple, but we need to introduce four more operators.

- The Kleene closure written * is used to iterate a process expression an arbitrary number of time.
- The choice which is denoted by $|$ and expresses that one of two process expressions can be executed.
- The quantified choice is a variant of the previous operator. When a process expression depends on the value a free variable, we can use a quantified choice to express the fact we can execute this process expression for a specific value of the free variable. We note it : $|x \in ens : \mathbf{a}(x)$, and it means that $\mathbf{a}(x)$ is executed with a specific value of x taken in the set *ens*.
- The wildcard _ is a syntax shortcut used instead of a quantified choice. As an example, $a(\_)$ means $|x \in ens : \mathbf{a}(x)$

In our work, we consider that security events are not necessarily correct : in fact we have in our modeling to verify that the user can for example play the role in the bank branch, both given by the security environment. Now, we introduce a process expression **permissionA** and explain it.

permissionA( ) $\triangleq$

$$
\begin{array}{ll}
(& \\
 & \langle agatha, banker, \_, \_, \mathsf{deposit}(\_, \_)\rangle \\
 \mid & \langle boris, \_, \_, \_, \mathsf{cancel}(\_, \_)\rangle \\
 \mid & \langle chris, clerk, toronto, \_, \mathsf{credit}(\_, \_)\rangle \\
)^* &
\end{array}
$$

This process expression, contains a Kleene closure over a choice of three actions. Thus, each event received must be compatible with one of this three action. The first action $\langle agatha, banker, \_, \_, \mathsf{deposit}(\_, \_)\rangle$ describes that if the event is an instance of deposit, thus the role mentioned in the security environment must be *banker* and the user must be *agatha*. This pattern of permission, is the same as those used in RBAC : permission to execute an action are given to a user playing a role. The second action $\langle boris, \_, \_, \_, \mathsf{cancel}(\_, \_)\rangle$ describes that if the event is an instance of cancel, thus

the person mentioned in the security environment must be *boris*. This pattern of permission is the same as those used in Bell et LaPadula [5] : permission are given to users. The third action $\langle chris, clerk, toronto, \_, \mathsf{credit}(\_,\_)\rangle$ describes that if the event is an instance of credit, thus the user, the role and the organization mentioned in the security environment must be respectively *chris*, *clerk* and *toronto*. This pattern of permission is the same as those used in OrBAC : permissions are given to a user playing a role in a bank branch.

This PE shows the different ways that can be used in an EB^{3}SEC modeling to express permission, as a matter of fact it does not depict our running example. Now, we give a new process expression called **permissionB**, that is a first PE that depict our running example. Going through the six rules of our running example, we only keep in mind the three first rules. In fact, those three rules are the only rules dealing with permissions. The three rules does not depend on the branch, but only on roles played by users during the action. The first rule states that each clerk and each banker must be allowed to perform the action deposit(.) The second rule states that each banker and chief agency must be allowed to perform actions validate(a)nd cancel(.) The third rule states that each banker and clerk are allowed to perform the action credit(.) We present those permissions, in PE called **permissionB**.

permissionB( ) $\triangleq$

$$
\begin{array}{l}
(\\
\quad \langle adrian, clerk, \_, \_, \mathsf{deposit}(\_,\_)\rangle\\
\quad \langle boris, banker, \_, \_, \mathsf{deposit}(\_,\_)\rangle\\
\quad | \ \langle boris, banker, \_, \_, \mathsf{cancel}(\_,\_)\rangle\\
\quad | \ \langle calvin, chief\ agency, \_, \_, \mathsf{cancel}(\_,\_)\rangle\\
\quad | \ \langle boris, banker, \_, \_, \mathsf{validate}(\_,\_)\rangle\\
\quad | \ \langle calvin, chief\ agency, \_, \_, \mathsf{validate}(\_,\_)\rangle\\
\quad | \ \langle boris, banker, \_, \_, \mathsf{credit}(\_,\_)\rangle\\
\quad | \ \langle adrian, clerk, \_, \_, \mathsf{credit}(\_,\_)\rangle\\
)^*
\end{array}
$$

In a sake of concision, we only instantiate one permission for each couple of role and action. As a fact, we assume that there is more than one user of each role, so we have to add the substantial number of lines for each user of the system.

In case studies provided by our industrial partner, we must keep in mind that the IS is used by a great number of users. Thus, it becomes quickly tedious to express permissions for each user in a process expression. In order to ease this task, EB^{3}SEC provides a class diagram. As a fact, for each security event received we have to check if it complies with at least one permission described in the modeling. PE are not designed for this kind of task, as they are no ordering constraints over actions. Thus the class diagram is used to defined class and relation useful to express permissions. So, for each security event received we have to check if it complies with permissions expressed in the class diagram.

In the following, we describe the kind of permissions we used in our example, and thus how we check the complying of security events with permissions described in the class diagram.

To describe the permission pattern used in our approach, we have have to keep in mind that they have to be easily used with case studies provided by our industrial partner. A first approach would have been to use a standard as RBAC [8] and used it. A first problem occurs, when we noticed that the AC policy derived from the case studies has to deal with a geographic context : for example the validation procedure of a check deposit differs from branches to branches if they are not grounded in the same state. Furthermore, in other part of the case studies permissions granted to a role are

different if the role is played in branches of different states. To ease the modeling of permission we used another AC model : OrBAC [**?**]. This model, includes the notion of *organization* in the pattern used to describe permissions. This pattern will be explained in the following.

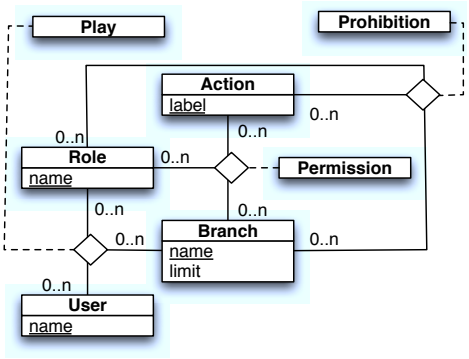

Figure 1: Class diagram used to express OrBAC style permissions

In this diagram we have four entities. Their instances correspond to users, roles, actions and branches used in our modeling. The relation *Play* describes the role associated with users in branches. The relation *Permission* expresses that permissions are given to roles in branches. This class diagram has to be instantiated to express permissions for the three first rules of our running example.

In a sake of readability, we used the name of each instance as a key. We keep in mind that this method allows the example to be more readable. The class *Branch* is instanced for each branch of the running example : Montreal and Toronto. The class *Action* is instantiated for each action aimed by the AC policy : deposit, cancel, validate and credit. The class *Role* is instantiated for each role that can be played in the IS : customer, clerk, banker and chief agency. The relation *Play* is instantiated to express role given to each role in a branch for an action. In our example, permissions are the same in both branch. But in case studies supplied by our partner, permissions differ from branches to branches. The class *User* is instantiated for each user of the system. The relation *Play* is instanced in order to expressed roles played users in branches.

Now, for each event received, we want to check if the security environment corresponds to the class diagram. To do so, we use a static predicate which is a first order logic predicate. And we have to check if this predicate holds for each event received.

$$
\begin{aligned}
&sp(p,r,o,t,e) = \\
&\qquad \langle p,r,o\rangle \in Play \\
&\wedge \quad \langle r,o,e\rangle \in Permission
\end{aligned}
$$

Permissions can be expressed by two ways : by using a class diagram or a process expression. Our experience shows that the class diagram is generally used to express permissions that fit well with the AC model express in the class diagram. Thus, process expression are used for permissions that are slightly exotic. For example, let suppose we have a superuser called *damien* that can execute the action *cancel* in both branches. To express this permission we could use this PE in addition to the class diagram :

$$
\begin{aligned}
&\textbf{permissionC}() \triangleq \\
&\quad (\\
&\qquad \langle damien, \_, \_, \_, \text{cancel}(\_,\_)\rangle
\end{aligned}
$$

User

name
adrien
boris
calvin
daria
elisa
franck

Role

name
customer
clerk
banker
chief agency

Action

name
deposit
cancel
validate
credit

Branch

name	limit
Montreal	10 000$
Toronto	8 000$

Play

User	Role	Organisation
adrian	clerk	Montreal
boris	banker	Montreal
calvin	chief agency	Montreal
daria	clerk	Toronto
elisa	banker	Toronto
franck	chief agency	Toronto

permission

role	branch	action
clerk	Montreal	deposit
clerk	Montreal	credit
banker	Montreal	deposit
banker	Montreal	cancel
banker	Montreal	validate
banker	Montreal	credit
chief agency	Montreal	cancel
chief agency	Montreal	validate
clerk	Toronto	deposit
clerk	Toronto	credit
banker	Toronto	deposit
banker	Toronto	cancel
banker	Toronto	validate
banker	Toronto	credit
chief agency	Toronto	cancel
chief agency	Toronto	validate

Figure 2: Instance of the class diagram used for the running example

$)^*$

As, we do not have any exotic rule in our example, we use the class diagram and the process expression used for permission, is empty :

permission$(\,) \triangleq$
$($
λ
$)^*$

Now, we use the same approach to show how EB^{3}SEC can be used to express prohibitions.

3.3 PROHIBITION - PROHI

When setting a prohibition, we want to express security environments that are not allowed to execute an action. To do so, we use the same approach as previously performed for permissions : we use a process expression, to describe the pattern of security environment prohibited for each action. In order to achieve, we must introduce a new operator.

- The guard denoted by $p \Longrightarrow a$ where p is a first order quantified logic predicate and a is a process expression, allows one to execute a only if the predicate p holds. If p does not hold, thus nothing is done.

Now, we introduce a process expression corresponding to prohibition, and we explain it in the following.

prohibitionA$(\,) \triangleq$
$($
$|\, p \in User.name() : p \neq elisa \Longrightarrow$
$\langle p, \_, \_, \_, \mathsf{cancel}(\_)\rangle$
$|\, o \in Branch.name() : o \neq Toronto \Longrightarrow$
$\langle \_, \_, o, \_, \mathsf{validate}(\_)\rangle$
$)^*$

This PE does not models our running example, it is only used to illustrate the concept of prohibition and the way they are modeled in EB^{3}SEC. In this process we use guards to avoid explicit values for some parameters of the security environment. In the first part, the prohibitions avoid the person called *elisa* to execute the action cancel in all possible cases. In the second part we avoid the action validate to be performed in Toronto.

Previously we have used the concept of permission to model the three first rules of our running example. In the following we use the concept of prohibition to make this modeling. As a fact for each role we have to express a prohibition for each action and role that are not explicitly written in the example. As for permission we first begin with a PE.

prohibitionB$(\,) \triangleq$
$($
$|\, r \in Role.name() : r \neq customer \Longrightarrow$
$\langle \_, r, \_, \_, \mathsf{deposit}(\_, \_)\rangle$
$\langle \_, r, \_, \_, \mathsf{cancel}(\_, \_)\rangle$
$\langle \_, r, \_, \_, \mathsf{validate}(\_, \_)\rangle$

prohibition

role	branch	action
customer	Montreal	deposit
customer	Montreal	cancel
customer	Montreal	validate
customer	Montreal	credit
clerk	Montreal	cancel
clerk	Montreal	validate
chief agency	Montreal	deposit
chief agency	Montreal	credit
customer	Toronto	deposit
customer	Toronto	cancel
customer	Toronto	validate
customer	Toronto	credit
clerk	Toronto	cancel
clerk	Toronto	validate
chief agency	Toronto	deposit
chief agency	Toronto	credit

Figure 3: Instance of the class diagram used for the running example

$$
\begin{array}{l}
\qquad\qquad\quad | \quad \langle \_, r, \_, \_, \mathsf{credit}(\_, \_) \rangle \\
\quad | \quad | r \in Role.name() : r \neq clerk \Longrightarrow \\
\qquad\qquad\quad \langle \_, r, \_, \_, \mathsf{cancel}(\_, \_) \rangle \\
\qquad\qquad\quad | \quad \langle \_, r, \_, \_, \mathsf{validate}(\_, \_) \rangle \\
\quad | \quad | r \in Role.name() : r \neq chiefagency \Longrightarrow \\
\qquad\qquad\quad \langle \_, r, \_, \_, \mathsf{deposit}(\_, \_) \rangle \\
\qquad\qquad\quad | \quad \langle \_, r, \_, \_, \mathsf{credit}(\_, \_) \rangle \\
)^*
\end{array}
$$

Still considering the fact that we are dealing with IS and a larger number of user, we choose to express prohibition with the class diagram (ref). As a fact we assume that a prohibition is defined for a role to execute an action in a branch. Here we finish to instantiate the class diagram by giving the instance of the class *Prohibition*.

Our goal is to enforce this prohibition expressed by a class diagram. To do so, we slightly change the static predicate in order to care about the relation *Prohibition*.

$$
\begin{array}{l}
sp(p, r, o, t, e) = \\
\qquad \langle p, r, o \rangle \in Play \\
\quad \wedge \quad \langle r, o, e \rangle \in Permission \\
\quad \wedge \quad \langle r, o, e \rangle \notin Prohibition
\end{array}
$$

Then the process expression corresponding to prohibition is transformed in the empty PE. In the system, we want that both permissions and prohibitions are applied simultaneously. Thus, we use a specific process expression called **main** that represents the global behaviour of the AC policy.

3.4 A FIRST VERSION OF THE MAIN PROCESS EXPRESSION - MAIN1

In the main expression, we want that all aspects (*i.e.*:permissions, prohibitions ...) are applied simultaneously to all event received by the IS. In the following we will introduce a first version of the main process expression. This version will allows permissions and prohibitions to be checked for each received event. To do so, we introduce a new operator.

- The parallel operator depicted as **a** $\|$ **b** allows one to execute two actions without specifying the order in which they have to be executed.

main( ) $\triangleq$
  permission( )
$\|$
  prohibition( )

In other words, we can say that the process expression **permission** and **prohibition** have to be executed, but they must synchronise on actions they have in common. We give there a few examples of events which are accepted or not, and the reason it is so.

- $\langle adrian, banker, Montreal, \mathsf{deposit}(zoe, 124)\rangle$ is not accepted : In fact, adrian is not allowed to be a banker in Montreal
- $\langle boris, banker, Montreal, \mathsf{deposit}(zoe, 24)\rangle$ is allowed : In fact it refers only to permission and not to prohibition, as it complies with permissions modeled, the security event is accepted.
- $\langle adrian, clerk, Montreal, \mathsf{deposit}(yves, 123)\rangle$ is allowed : In fact, it appears in **Permission** and **prohibition** and complies with both of them.

When an event is globally refused by **main**, the state of both process expression **permission** and **prohibition** does not change.

3.5 OBLIGATION

In AC policy, the notion of *obligation* is an organizational constraint that apply to user. As an example, we can express an obligation for a user to execute an action of acknowledgement after performing an action. Typically, it is the kind of constraint expressed in the sixth rule. Sometimes, the obligation takes effect between the user who performed the first action, and a member of a group which contains the first user for the performing of the second action.

In EB^{3}SEC, an obligation explains that two security events of the trace are linked together by the value of the requester of the security event. To express an obligation we use process expressions (PE) containing these operators in addition to operators previously explained.

- The sequence denoted by **a.b** allows one to execute the process expression **b** after the execution of the process expression **b**
- The interleaving denoted by **a** $|||$ **b** allows process expressions **a** and **b** to be executed in parallel without any synchronisation, even if they have common actions.
- The quantified interleaving is denoted by $|||\, p \in ens : \mathbf{exp}(p)$. It corresponds to the interleaving of process expressions **exp**(p) for all values of p contained in the set *ens*.

In our running example the sixth rule is an example of obligation. We introduce a new process expression that model this rule.

obligation( ) $\triangleq$
  $|||\, c \in User.name() :\ |||\, d \in int :\ |||\, m \in int :$
    $|\, p \in User.name() :$

$$(\quad \langle p,\_,\_,\_,\mathsf{deposit}(c,d,m)\rangle$$
$$\blacksquare\quad \langle p,\_,\_,\_,\mathsf{credit}(c,d,m)\rangle)^*$$

This PE models the sixth rules. In fact, as argument are explicitly written in deposit and credit, in the trace of the system, these actions are executed for particular value of a customer, a check number and an amount of a check. Now we suppose that the security event $\langle adrian, clerk, Montreal, \mathsf{deposit}(zoe,1,23)\rangle$ is already executed. Thus, a security event corresponding to the event $\mathsf{credit}(zoe,1,23)$ can not be performed unless the requester is adrian.

Our PE only express that the two actions have to be done by the same user. But as a fact the system can not physically force the user to perform the second action after executing the first one. A way to enforce the obligation is to forbid any action for the user after he executed the first one. We show another PE modeling obligation this way.

obligationP$() \triangleq$
$|||\, c \in User.name() :\ |||\, d \in int :\ |||\, m \in int :$
$|\, p \in User.name() :$
$(\quad \langle p,\_,\_,\_,\mathsf{deposit}(c,d,m)\rangle$
$\blacksquare\quad (\quad \langle p,\_,\_,\_,\mathsf{credit}(c,d,m)\rangle$
$|\ FALSE \Longrightarrow \langle p,\_,\_,\_,\_,\mathsf{deposit}(\_,\_,\_)\rangle$
$|\ FALSE \Longrightarrow \langle p,\_,\_,\_,\_,\mathsf{cancel}(\_,\_,\_)\rangle$
$|\ FALSE \Longrightarrow \langle p,\_,\_,\_,\_,\mathsf{validate}(\_,\_,\_)\rangle)^*$

In this PE, after performing the action deposit, the user have the choice between four actions. Three of this four actions are blocked by a guard that will never hold. Thus, the user can perform only the action deposit for the check deposit he performed. As it is not mentioned that other action are disabled during the obligation we use the first PE for our modeling.

3.6 A REFINEMENT OF THE MAIN PROCESS EXPRESSION

As we have described new constraints on our system, we have to integrate them in the main process expression that describes the global AC policy behaviour.

We put the process expression **obligation** in parallel with **permission** and **prohibition**.

main$() \triangleq$
permissions$()$
$||$
prohibition$()$
$||$
obligation$()$

3.7 SEPARATION OF DUTY

SoD (Separation of duty) is a security requirement that divide a task in subtasks and rely the execution of these subtasks to different users [7]. Thus, the security of the process has less chances to be corrupted by a single user who could execute all the subtasks by himself.

In RBAC, the problem of SoD is generally statically resolved : the permission of executing subtasks are given to different roles, and users are not allowed to play conflicting roles. We can use these approach in EB^{3}SEC by checking the instance of the class diagram.

Another approach used to solve SoD is called dynamic SoD. For two actions, with SoD constraints, dynamic SoD forbid the user to execute the second action, once the first action is executed. To achieve dynamic SoD, the method used must keep an instance of tasks already performed. In, EB^{3}SEC, instances of workflows and their executions are saved by the PE. This will be shown in the following.

Fourth and fifth rules deal with SoD constraints. We give in the following a PE modeling these constraints and then we explain them.

separation ( ) $\triangleq$

$|||\, c \in User.name() :\ |||\, d \in int :\ |||\, m \in int :$
$\quad |\, p \in User.name() :\ |\, p' \in User.name() :$
$\quad\quad (\quad \langle p, \_, \_, \_, \mathsf{deposit}(c,d,m)\rangle$
$\quad\quad \cdot\quad p \neq p' \Longrightarrow$
$\quad\quad\quad\quad \langle p', \_, \_, \_, \mathsf{validate}(c,d,m)\rangle$
$\quad\quad\quad\quad |\ \langle p', \_, \_, \_, \mathsf{cancel}(c,d,m)\rangle)^*$

$|||$

$|||\, c \in User.name() :\ |||\, d \in int :\ |||\, m \in int :$
$\quad |\, p \in User.name() :\ |\, o \in Branch.name() :\ |\, p' \in User.name() :$
$\quad\quad (\quad \langle p, \_, o, \_, \mathsf{deposit}(c,d,m)\rangle$
$\quad\quad \cdot\quad m > o.limit \Longrightarrow$
$\quad\quad\quad\quad p \neq p' \wedge p \neq p'' \wedge p' \neq p'' \Longrightarrow$
$\quad\quad\quad\quad \langle p', \_, \_, \_, \mathsf{validate}(c,d,m)\rangle$
$\quad\quad\quad\quad \|\ \langle p'', chiefagency, \_, \_, \mathsf{validate}(c,d,m)\rangle$
$\quad\quad\quad |\ m \leq o.limit \Longrightarrow \lambda$
$\quad\quad\quad\quad)^*$

In the first part of this PE, we expressed the fourth rule. We used a quantification over p and p' in order to achieve that the executer of deposit and validate or cancel must be different. Furthermore, we made a quantification over c, d and m to explicitly describe our workflow process. In fact, since the argument are explicitly written, the user who did the deposit for a check can still validate or cancel another check.

The second part of the PE models the fifth rule. For checks that value exceed the branch limit we must make two validate. Other wise the λ reminds that there is no special procedure to execute and only the fourth rule applies. The constraint used in this example for the check value is modeled in the PE by the expression $m > o.limit$ used in a guard. Furthermore, we still explicitly write the arguments of the actions in order to model workflows. Thus, the user who make a deposit can still make a cancel or a validate for another check.

In EB^{3}SEC, SoD constraints are defined at a workflow process level. As an example, for a user making the deposit of a check can still do other deposit but won't be able to make the validate or the cancel for the check for which he made the deposit. We illustrate this on the first part of the SoD : **separationP**.

separationP ( ) $\triangleq$

$|||\, c \in User.name() :\ |||\, d \in int :\ |||\, m \in int :$
$\quad |\, p \in User.name() :\ |\, p' \in User.name() :$

$$
\begin{array}{ll}
(& \langle p, \_, \_, \_, \mathsf{deposit}(c,d,m)\rangle \\
\blacksquare & p \neq p' \Longrightarrow \\
 & \quad \langle p', \_, \_, \_, \mathsf{validate}(c,d,m)\rangle \\
 & \quad | \ \langle p', \_, \_, \_, \mathsf{cancel}(c,d,m)\rangle)^*
\end{array}
$$

We suppose now we received the security event $\langle adrian, clerk, Montreal, 1234, \mathsf{deposit}(zoe, 1, 123)\rangle$. We apply the transition rules to **separationP** and show what it becomes :

separationP' $() \triangleq$

$$
\begin{array}{l}
|||\, c \in User.name()\backslash\{zoe\} \ : \ |||\, m \in float\backslash\{123\} \ : \\
\quad |\, p \in User.name() \ : \ |\, p' \in User.name() \ : \\
\quad\quad (\quad \langle p, \_, \_, \_, \mathsf{deposit}(c,m)\rangle \\
\quad\quad \blacksquare \quad p \neq p' \Longrightarrow \\
\quad\quad\quad \langle p', \_, \_, \_, \mathsf{validate}(c,m)\rangle \\
\quad\quad\quad | \ \langle p', \_, \_, \_, \mathsf{cancel}(c,m)\rangle)^* \\
|||\\
\quad ([c := zoe, d := 1, m := 123, u := adrian])|\, p \in User.name() \ : \ |\, p' \in User.name() \ : \\
\quad\quad (\quad \langle p, \_, \_, \_, \mathsf{deposit}(c,d,m)\rangle \\
\quad\quad \blacksquare \quad p \neq p' \Longrightarrow \\
\quad\quad\quad \langle p', \_, \_, \_, \mathsf{validate}(c,d,m)\rangle \\
\quad\quad\quad | \ \langle p', \_, \_, \_, \mathsf{cancel}(c,d,m)\rangle)^*
\end{array}
$$

This PE shows that, by now, the user adrian can not execute validate, neither cancel for the check deposit 1. But the user adrian can still do other check deposit and cancel or validate other check deposit.

3.8 A NEW REFINEMENT OF THE MAIN PROCESS EXPRESSION

As SoD is a component of our AC policy we want to incorporate it the main process expression. We achieve this by using a parallel with the older version of **main**

main $() \triangleq$

$$
\begin{array}{l}
\quad \textbf{permissions}() \\
\quad || \\
\quad \textbf{prohibition}() \\
\quad || \\
\quad \textbf{obligation}() \\
\quad || \\
\quad \textbf{separation}()
\end{array}
$$

4 DISCUSSION OF OUR MODELING

The first point we want to discuss about is the equivalence of using a PE or the class diagram to describe permissions and prohibitions. Prohibitions and permissions have to be checked for each security event received. Using PE or a static predicate with the class diagram are totally equivalent on a formal point of view. But, our experience shows that using the class diagram make the modeling

more readable for human. On an implementation point of view, our goal is to use an interpretor to execute PEs with an access to a database in order to keep an updated version of the instance of the class diagram. As we aim to secure IS involving large number of users, we think that using static predicates could be a way to optimize the execution of the interpreter. But PEs can still be used.

In this paper we described how to use permissions and prohibition. But we never force someone to use both in the same model. As permissions and prohibitions that can be found in requirements can be tedious to understand, we think that using both permissions and prohibitions can lead to deadlock.

The concept of *branch* used in our class diagram is very powerful on case studies provided by our partner. In the **permission** and **prohibition** it was not very useful, but it helped a lot when we had to model constraints over SoD.

The way we used to model the example, was directed by the fact we wanted to show how someone can model *permissions*, *prohibitions*, *obligations* and *SoD* in EB^{3}SEC. We also use another approach that consists to product a PE for each rule of the requirement.

Using PEs in EB^{3}SEC allows one to product a AC model that contains SoD and obligations rules, that are really accurate. As an example, we were able to express SoD rules with constraints and taking into account workflow processes.

5 IMPLEMENTATION

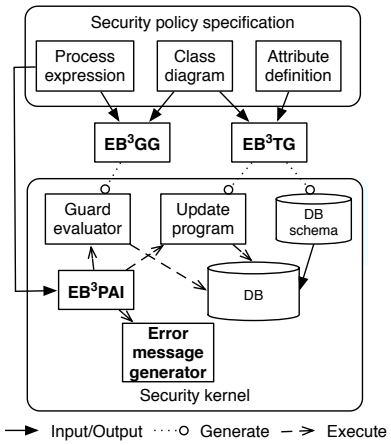

Figure 4: Architecture of the PDP

The EB^{3}SEC language uses the same process algebra than the EB^3 method, with adding the notion of security environment. EB^3 is implemented in a platform called APIS that allow its interpretation. Components of this platform can be reused to create a Policy Decision Point (PDP), which can be used in an SOA architecture. Figure 4 illustrates the architecture of the PDP. It is based on the APISplatform and reuses some of its components. EB^{3}PAI [9, 10] is an interpreter for EB^3 process expressions. It can be used to run EB^{3}SEC process expressions by translating security actions into normal actions. Process expressions can contain predicates referring to attributes of the requirements class diagram and the security class diagram. We suppose that each class diagram is implemented in its own relational database. To implement the security class diagram, we use EB^{3}TG [12] to generate a relational database schema and update programs to implement attribute definitions. Update programs are used by EB^{3}PAI to maintain the database in a consistent state while security events are executed. The module EB^{3}GG [17] is used to generate a *guard evaluator* program which contains procedures

that are used by EB^{3}PAI to determine if a guard holds. It generates SQL requests on the database to obtain attribute values of entities. These procedures can refer to security database and the IS database. If an event is refused by EB^{3}PAI, the *error message generator* [20] is used to generate an adapted error message.

EB^{3}PAI can execute arbitrary process expressions and uses an object-oriented database (OODB) to persistently store the state of process expressions. It can efficiently execute quantified expressions over arbitrary large sets (both choice and interleave) by using optimization techniques [9, 10]. Its main weaknesses are that its OODB is rather slow and its execution cannot be easily distributed over several processors. For high throughput banking applications, this is not sufficient. Hence, we are currently working on a new process algebra interpreter which will be restricted to deterministic and optimizable process expressions. These process expressions can be translated into an algebraic state transition diagram (ASTD) [11]. The execution of an ASTD is easier to distribute over several processors, thereby exploiting parallelism in the processing of events. The state of an ASTD is more compact than the one used in EB^{3}PAI and can be easily stored into a relational database. The use of ASTDs will also enable the use of state-based techniques like B and Event-B to prove properties about a security policy.

6 FUTURE WORK

The next step in our work will be to include the capabilities of making model checking over EB^{3}SEC models. As EB^{3}SEC is very similar to EB^3 we plan to reuse results and tools developed for EB^3. As a result, Alloy will surely be the tools we will use. The first step will be to transform EB^{3}SEC models into Alloy model. The first step will be to define how to transform the static part of an EB^{3}SEC model (i.e. : class digram, permissions and prohibitions) into an alloy model. At this step we will be able to verify the consistency between permissions rules and prohibition rules. After, we will need to transform the dynamic part (i.e. : obligation, SoD and more generally all PEs) in an Alloy model. Thus, we will be able to verify that dynamic and static parts are coherent.

Furthermore, if the functional part of the IS is also formally modeled we will be able to check the coherence between the functionnal and the security parts.

The implementation previously presented is also a fuure work. In order to integrate tools of the APIS platform for EB^{3}SEC we plan to develop a prototype made in OCaml and SQL. In the project lead with our industrial partner, we plan to implement a platform based on EB^{3}SEC and usable in a real life platform. This platform has to be integrated in an SOA architecture.

7 RELATED WORK

In order to compare related work, we use the approach describe in [1]. This approach describes a framework that allows one to express characteristics of AC model. This approach leads to thirty different comparison points encompassed in eight categories. As we present here only the EB^{3}SEC language and not the full EB^{3}SEC project led with our industrial partner, we only mention a few comparison points and categories. We apply this approach to our work and then to other related work:

- In [3], the authors show a method aiming to solve AC problems by including a RBAC model in the UML design design of the application
- In [4], the authors show a method helping to create a language to express SoD express in CSP with an RBAC model in order to be used with an enforcement point

- [21] proposes a non formal language to express AC policies. In fact the problem is that contradictory decision can be taken for a request by the same policy. Several works as [16] presents a formalisation of policies expressed in XACML in order to avoid the problem. [2] presents a profile made in XACML of the RBAC model including mechanisms to use SoD. In the following, we consider this set of works as a unique work called XACML considered as formal and able to express SoD. Here, the SoD is used for a session during which a user can not trigger conflicting role in the same time.

Now, we deal with the few comparison points from (ref ORKA).

7.1 model specification

The first point to consider in this category is the level of abstraction that can be used in the model. With EB^{3}SEC or [4], we deal with a high level of abstraction, for example in a SOA architecture we deal with web-services. In [3] amd XACML, we deal with object and operation that are implemented in the code of the software.

In [4], XACML and EB^{3}SEC the underlying model of the specification is formal. However if we consider the XACML with only the standard it is non formal whereas [3] is structured and semi-formal; it is structured as it is using schemas and UML notation and semi-formal as it is using for example first order predicates to express constraints for permissions.

All of these four method have the same purposed : they are used to enforce a AC policy in a IS. They also tends to have the same using area : IS built on a SOA architecture.

In [4], [3] and XACML the underlying model of the specification is a RBAC model. In other word, AC authorizations have to comply with the RBAC model. In EB^{3}SEC, the underlying model, is chosen by the person who models the policy. In this paper, we choose a model derived from the OrBAC methods, but in other cases we could use a RBAC model or a home made model designed to ease the modeling step.

7.2 Policy expressiveness

[3] allows one to express permissions with constraints on the model. [4] allows one to express permission and SoD constraints at the workflow level. XACML allows one to express permissions, prohibitions and Sod constraints on the model. EB^{3}SEC allows one to to express permissions and prohibitions with constraints, obligations and Sod at a workflow level on the model.

In EB^{3}SEC and [4] the model includes workflow routing concepts such as serialization, selection, iteration, and parallelization of steps and is able to control the order of events in a fine-grained manner.

In this paper we present only the EB^{3}SEC language. Thus, we do not mention the other comparison point of [1] which deals with our future work or with other parts of our project. Compared to the three other methods, EB^{3}SEC seems to be a language more powerful in term of expressiveness : In fact, it had the particularity of allowing the modeler to use a standard model or a home made model and also to allow him to express different constraints as permissions, prohibitions, obligations and SoD on a finned grained level. All of these constraints are not allowed at the same time by a unique method at a time.

References

[1] Christopher Alm, Michael Drouineaud, Ute Faltin, Karsten Sohr, and Ruben Wolf. A classification framework designed for advanced role-based access control models and mechanisms. Technical report, Technologie-Zentrum Informatik Bremen University, 2009.

[2] A Anderson. XACML Profile for Role Based Access Control (RBAC). OASIS Standard, 2004.

[3] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From uml models to access control infrastructures. ACM Trans. Softw. Eng. Methodol., 15(1):39–91, 2006.

[4] David A. Basin, Samuel J. Burri, and Günter Karjoth. Dynamic enforcement of abstract separation of duty constraints. In Michael Backes and Peng Ning, editors, ESORICS, volume 5789 of Lecture Notes in Computer Science, pages 250–267. Springer, 2009.

[5] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations and model. reconstruction électronique 2547 v2, MITRE Corporation, 1973.

[6] A. Abou El Kalam *et al*. Organization based access control. In 4th Intl. IEEE Workshop Policies for Distributed Systems and Networks, pages 120–130, Como, Italy, 2003. IEEE Press.

[7] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. Role-Based Access Control. Artech House, Inc., Norwood, MA, USA, 2003.

[8] American National Standard Institute (ANSI) for Information Technology. Role Based Access Control. ANSI INCITS 359-2004, februar 2004.

[9] B. Fraikin and M. Frappier. Efficient symbolic execution of large quantifications in a process algebra. In Jim Woodcock and Jin Song Dong, editors, ICFEM 2007, volume 4789 of LNCS, pages 327–344. Springer Berlin/Heidelberg, November 2007.

[10] B. Fraikin and M. Frappier. Efficient symbolic computation of process expressions. Science of Computer Programming, 74(9):723–753, July 2009.

[11] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. St-Denis. Extending statecharts with process algebra operators. In Innovations in Systems and Software Engineering, pages 285–292, London, UK, august 2008. Springer London.

[12] F. Gervais, M. Frappier, and R. Laleau. Generating relational database transactions from EB^3 attribute definitions. 8(3):423–445, July 2009.

[13] Camadian Government. Personal information protection and electronic documents act. Canadian Law, April 2000.

[14] Carlos Gutiérrez, Eduardo Fernández-Medina, and Mario Piattini. Towards a process for web services security. Journal of Research and Practice in Information Technology, 38(1), 2006.

[15] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

[16] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In WWW '07: Proceedings of the 16th international conference on World Wide Web, pages 677–686, New York, NY, USA, 2007. ACM.

[17] P. Konopacki. Synthèse automatique de guarde EB3. Master's thesis, Université de Sherbrooke, 2008.

[18] Peng Liu and Zhong Chen. An access control model for web services in business process. In WI '04: Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence, pages 292–298, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Francis Mer. loi de sécurité financière. Journal Officiel, (177), january 2003.

[20] J. Milhau, B. Fraikin, and M. Frappier. Automatic Generation of Error Messages for the Symbolic Execution of EB^3 Process Expressions. In Integrated Formal Methods: 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009, Proceedings, volume 5423 de LNCS, pages 337–351. Springer Berlin/Heidelberg, 2009.

[21] T Moses. eXtensible Access Control Markup Langage (XACML) Version 2.0. OASIS Standard, 2005.

[22] R. Sandhu. Transaction control expressions for separation of duty. In 4th Aerospace Computer Security Application Conference, pages 282–286, 1988.

[23] Paul Sarbanes and Mike Oxley. Sarbanes-oxley act. Public Law, (116):107–204, 2002.

[24] Emin Gün Sirer and Ke Wang. An access control language for web services. In SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies, pages 23–30, New York, NY, USA, 2002. ACM.