ANR programme ARPEGE 2008

Systèmes embarqués et Grandes Infrastructures

Projet SELKIS : Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

ANR-08-SEGI-018

Février 2009 - Décembre 2011

Spécification des règles de traduction

Livrable 5.1 bis

Jérémy Milhau
Michel Embe Jiague
Régine Laleau
Frédéric Gervais

LACL

Novembre 2011

Table des matières

Première partie

Introduction

Dans ce livrable nous présentons les résultats de deux approches de traduction du PIM d'une spécification de politique de contrôle d'accès en PSM. Les méta-modèles PIM et PSM de la politique de contrôle d'accès utilisés dans ce livrable ont été présentés dans le livrable 5.1. Dans une premier temps nous présentons dans la partie II une traduction PIM-PSM basée sur la traduction d'une spécification ASTD en BPEL. Puis nous présentons dans la partie III une approche de PIM combinant ASTD, SecureUML et UML en B qui regroupe ainsi les modèles statiques et dynamiques de la politique de contrôle d'accès avec les aspects fonctionnels du système. Le tout est traduit en B suivant un méta-modèle précis, puis raffiné selon une stratégie qui permet d'obtenir un PSM proche de l'implémentation finale de la politique.

Deuxième partie

ASTD vers BPEL

Chapitre 1

Application d'une politique de contrôle d'accès ASTD avec des processus WS-BPEL dans un environnement SOA

Le contrôle des accès dans le cadre de service web d'agences publiques ou d'entreprises privées dépend principalement de la capacité des spécification à satisfaire les règles et les normes définies par les gouvernements et les lois. Ceci est particulièrement vrai dans le cadre des systèmes bancaires et médicaux. Cet article présente les règles de transformation automatique de spécifications de politiques de contrôle d'accès en processus WS-BPEL. Cet spécification est décrite en utilisant des patrons de règles de contrôle d'accès utilisant la notation ASTD. Les processus BPEL ainsi générés sont exécutés par un moteur BPEL intégré dans une point de décision qui est un composant d'un *policy enforcement manager* (PEM) similaire à ceux proposés par le standard XACML.

Cet article a été soumis et accepté dans le journal *International Journal of Systems and Service-Oriented Engineering (IJSSOE).*

Enforcing ASTD Access-Control Policies with WS-BPEL Processes in SOA Environments

Michel Embe Jiague,[1,2] Marc Frappier,[1] Frédéric Gervais,[2]
Régine Laleau,[2] and Richard St-Denis[1]

[1]GRIL, Département d'informatique
Université de Sherbrooke
Sherbrooke (Québec), J1K 2R1, Canada
{Michel.Embe.Jiague,Marc.Frappier,Richard.St-Denis}@USherbrooke.ca

[2]Université Paris-Est Créteil Val-de-Marne, LACL, IUT Sénart Fontainebleau
Département Informatique, Route Hurtault, 77 300 Fontainebleau, France
{Frederic.Gervais,Laleau}@U-Pec.fr

Abstract

Controlling access to the Web services of public agencies and private corporations depends primarily on specifying and deploying functional security rules to satisfy strict regulations imposed by governments, particularly in the financial and health sectors. This paper focuses on one aspect of the SELKIS and EB^{3}SEC projects related to the security of Web-based information systems, namely, the automatic transformation of security rules into WS-BPEL (or BPEL, for short) processes. The former are instantiated from security-rule patterns written in a graphical notation, called ASTD, that is close to statecharts. The latter are executed by a BPEL engine integrated into a policy decision point, which is a component of a policy enforcement manager similar to that proposed in the XACML standard.

Keywords: Access-control policy, security rule, policy decision point, ASTD, EB^{3}SEC, BPEL, transformation, SOA.

Introduction

In some business sectors, information systems (IS) are governed by internal organization policies and government laws. Access control is widely used in IS to enforce such policies as well as to prevent breaches of data confidentiality and integrity security. More precisely, user access to data and functionalities are filtered, based on well-defined policies. Role-based access control (RBAC), a methodology that associates user identities with the data and/or functionalities through their role, is the most commonly implemented solution. It does not, however, solve new problems found in today's common service-oriented architecture (SOA) applications. These applications are not solely "one user centric" and implement workflows that may involve interactions with different users. With respect to workflows, RBAC has little expressiveness power. As an example, RBAC frameworks cannot implement separation of duty (SoD) properties.

A substantial part of the EB^{3}SEC (EB^{3}SEC stands for EB^{3}Secure) and SELKIS (SELKIS is an acronym for SEcure heaLth care networKs Information Systems) projects consists in developing a prototype of a policy enforcement manager (PEM) for distributed IS executed in an SOA environment as Web services (WS). Our approach focuses on three levels of access control: the data level, the RBAC level, and the process level. In this paper, we propose an automatic implementation of a significant part of the enforcement framework derived from an access-control policy expressed in a high-level language. This high-level language is formal, powerful enough to implement common properties encountered in security policies, and can also express many sorts of constraints. Its implementation relies on a translation algorithm, which produces an executable BPEL process from a formal specification of an access-control policy. Overall, the enforcement framework follows architectural guidelines proposed in the XACML standard.

The rest of this paper is organized as follows. The next section introduces Algebraic State Transition Diagram (ASTD), the formal notation used for specifying security rules and presents four security-rule patterns: permission, prohibition, obligation, and SoD. The ASTD notation provides for combining state-transition diagrams with process algebra operators. This high-level notation is appropriate for specifying security rules at the process level and is independent of any implementation environment. The following section presents the architecture of SOA applications targeted by our projects and the three levels of granularity we identified previously. Enforcement framework components are described and typical message-exchange scenarios between the principal components are depicted. The sequel details a translation schema that transforms an ASTD specification into a BPEL process along with its WSDL interface and XSD type definitions, which are deployed in an SOA environment so that they constitute the core of the policy decision point (PDP). This translation schema is mechanizable as far as the security rules obey the aforementioned patterns. A safe translation procedure then replaces an error-prone development phase. The last sections describe strongly related work and point out differences with some aspects developed in this paper and conclude with ongoing and future aspects of this work.

Expressing Security Rules with an ASTD

In most security frameworks, a security policy is a combination of many security rules. Researchers and security practitioners (Basin, Burri & Karjoth, 2009; Konopacki, Frappier & Laleau, 2010a; Konopacki, Frappier & Laleau, 2010b; Yao, Moody & Bacon, 2001) have considered the following categories for security rules:

- permission which authorizes actions to be executed;
- prohibition which forbids actions to be executed;
- separation of duty which expresses the fact that a set of tasks cannot be executed by the same users or roles;
- obligation which forces a user to perform an action sometime in the future after he has performed a specific action. In other words, two distinct actions must be performed by the same user.

Such rules can be expressed with ASTD, which is a graphical and formal notation initially created to design IS. It has been inspired from statecharts (Harel, 1987) and process algebras like CSP (Hoare, 1978) and LOTOS (Bolognesi & Brinksma, 1987). Since ASTD notation is formal, a wide range of verification and validation tools can be used with ASTD-based models. In particular, Milhau et al. (2010) devised a transformation from ASTD to Event-B (Abrial, 2010), a formal language which has robust, well-established tools for proofs and formal verification. ASTD notation can also be used to define security rules that restrict the free behavior of an IS so that it does not violate organizational access-control policies.

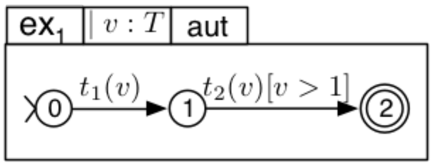


Figure 1. First example

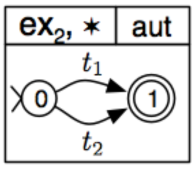


Figure 2. Second example

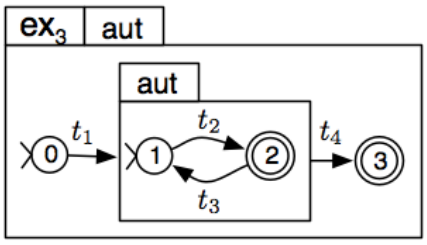


Figure 3. Third example

An ASTD—the building block of an ASTD specification—is a hierarchical transition system. The dynamic is based on transitions labeled with events in the form t(**x**)[ϕ], where **x** is a list of parameters (which can be empty) and ϕ is an optional guard that must hold to enable the transition to fire. States are typed, with possible types including elementary state, automaton, guard, and choice. A special feature of ASTD notation is a quantified version of parameterized synchronization and choice operators. In addition, each non-elementary state may carry parameters as events do. Figure 1 shows a quantified choice that sequentially executes event t_1 for any value of $v \in T$ and t_2 for a previously accepted value of v that complies with the predicate $v>1$. Figure 2 shows a Kleene closure (∗) in which either event t_1 or event t_2 will be executed repetitively. The third example in Figure 3 illustrates usage of hierarchy in an automaton to encapsulate a complex behavior in a macro state. The complete definition of ASTD notation and its formal operational semantics is available in (Frappier, Gervais, Laleau & Fraikin, 2008). Concerning expression of security policies with ASTD, transitions are augmented with two parameters to take into account the user's identity and role while executing an action: $\langle u,r \rangle$ where u is the user identifier and r his role. Some applications may consider other parameters relevant to security rules, such as organization or time.

A Simple Use Case: The Bank

In this section, a simple banking example is described to illustrate expression of security rules using ASTD notation and the translation of these rules into an executable BPEL process. This case is a simplification of a real case provided by one of our industrial partners. The example concerns depositing a check in a bank. The check is brought to the bank by the `client`, who does not have access to the bank's IS. It is then deposited in the client's account at the bank either by a `cashier` or a `head office`. As stipulated by company policy, the bank's IS updates the client's account balance after the deposit is validated by a `head office`. To prevent possible IS misuse and safeguard the bank's data consistency, a security policy that comprises four rules is defined:

Rule 1Action `deposit` is permitted for both roles `cashier` and `head office`, while action `cancel` is restricted to the role `head office`.

Rule 2Only `head offices` can validate or cancel a check deposit.

Rule 3For discretion reasons, a client is registered with a `cashier` and only this `cashier` or a `head office` is allowed to make deposits on the client's behalf.

Rule 4When entered by a `cashier`, a deposit is validated or cancelled by a `head office`. Otherwise, a `head office` made the deposit and it must be validated or cancelled by another `head office`.

ASTD security rule examples

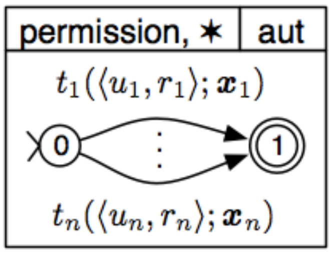

permission, ∗
aut
$t_1(\langle u_1, r_1\rangle; \boldsymbol{x}_1)$
0
1
$t_n(\langle u_n, r_n\rangle; \boldsymbol{x}_n)$

Figure 4. Permission pattern

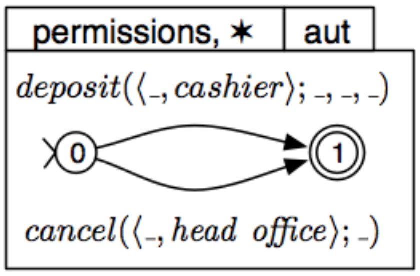

permissions, ∗
aut
$deposit(\langle \_, cashier\rangle; \_, \_, \_)$
0
1
$cancel(\langle \_, head\ office\rangle; \_)$

Figure 5. Permission instance

Figure 4 shows the ASTD pattern for permission. It enables the execution of action t_1 (to t_n) with parameters $\mathbf{x}_1$ (to $\mathbf{x}_n$) by user u_1 (to u_n) acting with role r_1 (to r_n). A Kleene closure (∗) is used, since an action can be executed repeatedly. Figure 5 illustrates an instance of this pattern in which roles `cashier` and `head office` are given permissions to execute actions `deposit` and `cancel` respectively. The symbol "-" denotes a "don't care" value for a parameter (account number, check's number and amount in the case of a deposit). In this instance of the pattern permission, the user is not specified for both actions `deposit` and `cancel` and so the rule does not filter on the user executing the action.

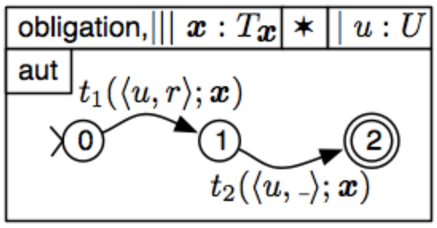

obligation, ||| $\boldsymbol{x} : T_{\boldsymbol{x}}$
∗
| $u : U$
aut
$t_1(\langle u, r\rangle; \boldsymbol{x})$
0
1
2
$t_2(\langle u, \_\rangle; \boldsymbol{x})$

Figure 6. Obligation pattern

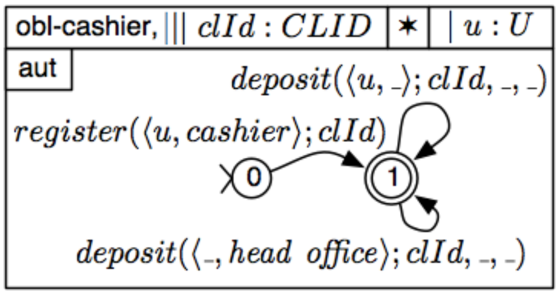

obl-cashier, ||| $clId : CLID$
∗
| $u : U$
aut
$deposit(\langle u, \_\rangle; clId, \_, \_)$
$register(\langle u, cashier\rangle; clId)$
0
1
$deposit(\langle \_, head\ office\rangle; clId, \_, \_)$

Figure 7. Obligation instance

Figure 6 includes the ASTD pattern for obligation. With such a rule, actions t_1 and t_2 must be executed by the same user u. As illustrated in Figure 7, deposits from a client must be always executed by the same `cashier`, unless the operation is done by the `head office`.

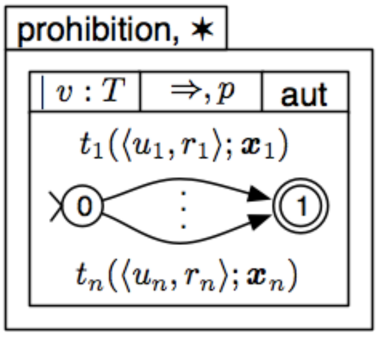


Figure 8. Prohibition pattern

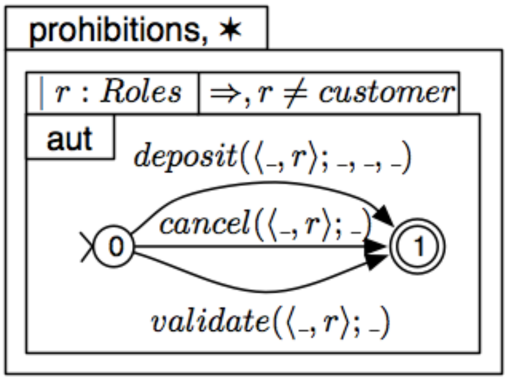


Figure 9. Prohibition instance

As mentioned before, prohibition forbids the execution of an action to a user or a role. The ASTD pattern for prohibition is depicted in Figure 8: actions t_1 to t_n are allowed to execute only when the predicate p holds. Figure 9 depicts an instance of this pattern and implements the part of the bank's policy that prohibits the `customer` from accessing the system. The guard predicate restrains the valid roles for actions `deposit`, `cancel`, and `validate`. Figure 10 includes the SoD pattern. Actions t_1 and t_2 are executed sequentially and the predicate u≠u' requires them to be executed by two different users. A more complex criterion might be specified, for example, action t_2 might be specified to be executed by a hierarchical superior of the user u who executed action t_1. Figure 11 illustrates a simple SoD constraint, which is an instance of the SoD pattern. It states that actions `deposit` and `validate` must be executed by two different users (u≠u').

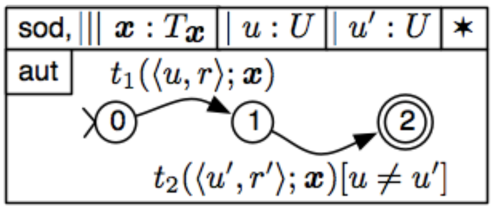


Figure 10. SoD pattern

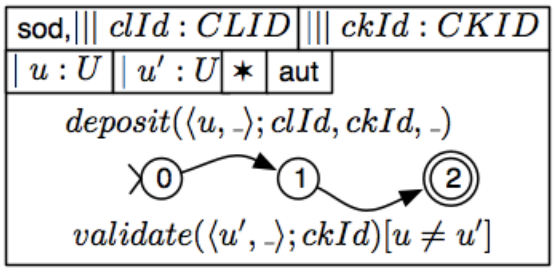

sod, ||| clId : CLID
||| ckId : CKID
| u : U
| u′ : U
∗
aut
deposit(⟨u, _⟩; clId, ckId, _)
0
1
2
validate(⟨u′, _⟩; ckId)[u ≠ u′]

Figure 11. SoD instance

It should be noted that limiting security rules to patterns makes it possible to derive BPEL processes from these particular forms of rules. Indeed, automatic translation regardless of ASTD structure represents a complex and challenging problem that exceeds the scope of our projects.

Architecture and Target Platform

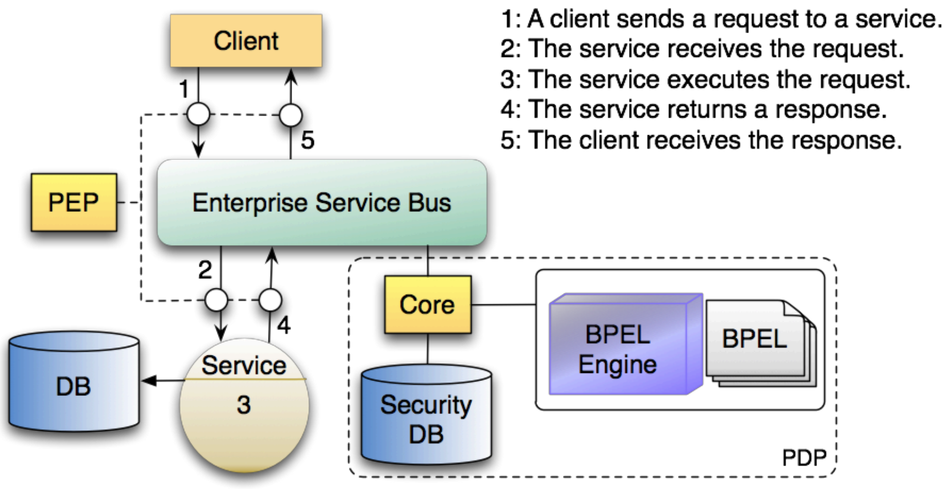

Client
1
5
PEP
Enterprise Service Bus
2
4
DB
Service
3
Core
Security DB
BPEL Engine
BPEL
PDP
1: A client sends a request to a service.
2: The service receives the request.
3: The service executes the request.
4: The service returns a response.
5: The client receives the response.

Figure 12. A typical SOA application

Figure 12 depicts a typical IS and its interaction with a client application. In this particular view, the client application sends a request to a WS using standard protocols such as HTTP, HTTPS (secured HTTP), WSDL, and SOAP. The request goes through an enterprise service bus (ESB) acting as middleware for the environment and a routing point for secure exchanges of messages between communicating partners. In our projects, the PEM complies with that specified in the XACML standard from OASIS (2005). It is based on two main components: the policy enforcement point (PEP) and PDP. Together, they are responsible for intercepting requests from client applications to services and providing authorization control with respect to access decisions for these requests based on security policies. There are two other auxiliary components to consider: the policy administration point (PAP), which provides facilities for the management of a policy repository, and the policy information point (PIP), which supplies additional information closely related to requests (e.g., roles, actions/services, environment) when required by the PDP.

The PDP plays a key role in the PEM, since it makes approval/denial decisions based on security policies. To ensure security with a high level of granularity, decisions are based on three different levels of functional security, as shown in Figure 13. The work described in this paper focuses on enforcement of functional security rules associated with the third level, called the process level. The reader is referred to a companion paper for a presentation of aspects related to the two other levels (Embe Jiague, Frappier, Gervais, Konopacki, Laleau, Milhau & St-Denis, 2010). Functional security rules defined at the third level concern business processes (collections of related structured atomic services). They describe rules that depend on the system state (e.g., the history of past events accepted by the system) and are specified at an abstract level using ASTD (Frappier, Gervais, Laleau, Fraikin & St-Denis, 2008). Generally, an ASTD takes into consideration a set of security rules that defines an access-control policy. Indeed, the security rules are put together in the same ASTD with the parallel composition operator. At the implementation level, the decision-making task is performed with a BPEL engine that enforces security rules from a BPEL process derived from an ASTD. Therefore, the rules are not attached to actions or services to secure, nor to entities (e.g., roles, actors) involved in the IS. For example, a specialist can consult a patient's health record only when the patient has been referred to him by the treating physician.

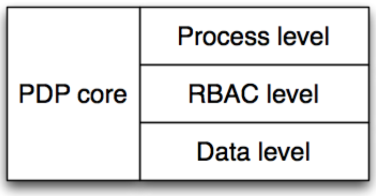


Figure 13. PDP abstract internal view

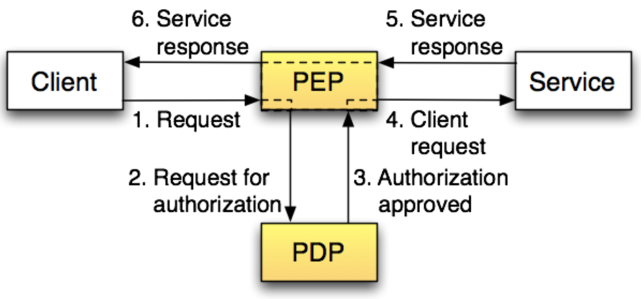


Figure 14. PEP and PDP security schema

Figure 14 details the interaction between a client and a service through the PEP as well as the interaction between the PEP and the PDP. In a typical scenario, a client sends a request to a service or a component of a distributed application (1) along with some user information (identifier and role). The request is intercepted by the PEP, which extracts user information and then formulates an authorization request for approval/denial by the PDP (2). The PDP makes a decision on whether to approve or deny the client request (in this scenario, the request has been approved by the PDP). This decision, centralized at the PDP core component, is based on checking user information (identifiers and roles in security DB) and the response from the BPEL engine to a specific request formulated by the core. The authorization decision is reported back to the PEP (3). If the PDP allows the request, then the PEP permits the original request to reach the requested service (4), which may perform specific business validations before executing the request (e.g., checking that an account has sufficient funds before initiating an electronic transfer). The response goes through the PEP (5), so that the policy repository or PAP (if any) can be updated on the recently executed request. Finally, the response is redirected to the client (6). PDP request denial is similar, except that steps (4) and (5) are superfluous, since the PEP immediately returns an authorization denial upon PDP rejection. In both cases, messages must be sent through secure channels to guarantee communication confidentiality and integrity to all partners. It should be noted that this schema is a simplification of the security data-flow diagram described in the XACML standard.

Transforming an ASTD Access-Control Specification into a BPEL Process

The algorithm that translates an ASTD access-control specification into a BPEL process is recursive and is described in the following subsections with a pattern-matching style for the function parameters. The algorithm generates a BPEL process along with its WSDL interface and XSD type definitions. The WSDL interface, required by the BPEL engine, describes the BPEL process as a WS (its operations and parameters). The XSD-type document describes the parameter types. The transformation is event-based in the sense that the BPEL process mimics the flow of events described by the ASTD.

The BPEL Process Language

WS-BPEL stands for Web Service Business Process Execution Language. It is basically an XML language for designing business processes independently from enactment engines. BPEL is an OASIS standard (OASIS, 2007) and makes intensive use of other standards. Because tasks are often automated through WS, it uses WSDL (the standard for describing service interfaces) and XPath, for navigation through XML variables.

A BPEL specification defines a process through the XML root element **process.** Such a process can be directly run by an enactment engine, so it is referred to as an executable process. BPEL provides various basic activities, such as message arrival (element **receive**), message reply (element **reply**), and WS invocation (element **invoke**). The standard includes more elaborated constructs, such as scopes (**scope**) for variable declaration, loops (e.g., **repeatUntil**), and conditionals (e.g., **if**). Furthermore, complex parallel processing is possible with the **flow** element, combined with **link** to create dependencies or synchronization points between concurrently running activities.

Our industrial partners in Canada and France are replacing their legacy systems using SOA platforms. As BPEL integrates very well into such environments, it is an appropriate choice for implementing security processes. Since engines deployed in a distributed environment are bundled with technical security characteristics (e.g., encryption protocols, reliable messaging, scalability), any proper BPEL engine would provide such functionalities at no additional cost. Another motivation behind the choice of BPEL is the work done to verify properties on BPEL processes (Aït-Sadoune & Aït-Ameur, 2010), which allows for checking properties on the final process using Event-B (Abrial, 2010) and the Rodin platform (Abrial, Butler, Hallerstede, Hoang, Mehta & Voisin, 2010).

Transformation Rules: ASTD to BPEL

The proposed transformation is tailored for the security-rule patterns expressed with ASTD notation. Each pattern is transformed into a set of adequate BPEL elements by processing each ASTD in the specification individually. This section provides a skeleton overview of these BPEL elements for the four patterns.

```
<repeatUntil>
 <pick>
  <onMessage operation="authDect1"…>
   …
  </onMessage>
  …
  <onMessage operation="authDectn"…>
   …
  </onMessage>
 </pick>
 <condition>false()</condition>
</repeatUntil>
```

Figure 15. Permission/prohibition BPEL code skeleton

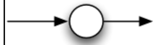
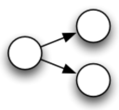
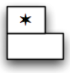
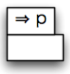
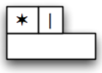
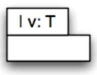
```
<repeatUntil>
 <sequence>
  <scope>
    <!-- t1 BPEL code -->
  </scope>
  <scope>
    <!-- t2 BPEL code -->
  </scope>
 </sequence>
 <condition>false()</condition>
</repeatUntil>
```

Figure 16. Obligation/SoD BPEL code skeleton

Intuitively, permission and prohibition patterns are implemented by a **repeatUntil** BPEL activity. As shown in Figure 4 and Figure 8, the events t_1 to t_n have the same outgoing state 0, thus introducing a **pick** activity (see Figure 15). The interaction between the PEP and PDP regarding a permission can be summarized as follows: When the part of the BPEL code corresponding to the pattern is tested, a request in the form authDect(⟨u,r⟩,**x**) (where t is the event, u and r the user and role, respectively, and **x** the event's parameters) is received by the **pick** activity. If $t \notin \{t_1, \ldots, t_n\}$, then the request is immediately denied, since the **pick** element cannot process it. Otherwise, the processing goes on to the corresponding **onMessage** element, where the user identity and role as well as event's parameters are tested against the values encoded in the BPEL code from the formal specification. Whether or not the match fails, the test will loop due to the upper **repeatUntil** element. The same applies to a prohibition rule except that, in addition to matching user and role against specification values, the predicate p (see Figure 8) is also evaluated as part of the authorization decision.

Similarly, obligation and SoD patterns are implemented by a **repeatUntil** BPEL element. The events t_1 and t_2 in Figure 6 and Figure 10 are transformed into a set of elements wrapped in a **sequence** BPEL element. Figure 16 depicts a skeleton of the transformation. As is the case with both permission and prohibition patterns, the BPEL code associated with obligation and SoD patterns follows the same layout with a noticeable difference due to the predicate introducing the “separation of duty”. Table 1 summarizes the mappings used to guide the transformation rules.

Table 1. Mapping between ASTD and BPEL constructs

	ASTD	BPEL
Sequence in automaton		<**sequence**/>
Choice in automaton		<**flow**/> with <**link**/> to manage dependencies between the first action and the remainder of each branch
Kleene closure		<**repeatUntil**/> with condition set to false
Guard		Add the predicate p to all the first possible events in the sub-ASTD
Prohibition choice		<**flow**/>
Quantified choice		<**scope**/> with two <**variable**/> based on v to manage the choice

Transformation Algorithm

The algorithm’s input is a complete ASTD specification, i.e., the main ASTD and the event signatures. In Figure 17, t^1 to t^r are the events used in the ASTD and elements p^i_j are the parameters of event t^i, T^i_j being the parameter types.

$$spec \leftarrow \langle main, a_1, \ldots, a_m; \langle t^1, \langle p^1_1, T^1_1 \rangle, \ldots, \langle p^1_{n1}, T^1_{n1} \rangle \rangle, \ldots, \langle t^r, \langle p^r_1, T^r_1 \rangle, \ldots, \langle p^r_{nr}, T^r_{nr} \rangle \rangle \rangle$$

Figure 17. The ASTD specification

The transformation algorithm is scattered across multiple functions. Functions T^{BPEL}, T^{WSDL}_{spec}, and T^{XSD}_{spec} produce, respectively, the BPEL process code, the WSDL definitions, and the XSD types required to effectively enact a security policy. The function that translates the ASTD specification into BPEL code uses a context Γ to move relevant information from the outer ASTD to the inner. This context comprises the following fields that help with computation:

- Γ_{corr} contains a list of pairs (x, init), where x is a quantified variable (from a quantified ASTD or artificial variable) for which BPEL correlation elements will be created and init a Boolean value (true or false), which tells the algorithm whether or not to initialize x’s associated correlation.

- Γ_{ch} contains the list of variables from quantified choices ASTD and is used to create appropriate BPEL variables to mimic the choice during BPEL process execution.
- Γ_g is a conjunction of guard predicates. It is a means to move the predicate from the parent guard ASTD to first possible event in the compound ASTD (see Figure 21).

```
$T^{BPEL}$($spec$):
  $\Gamma \leftarrow (\{\}_{corr}, \{\}_{ch}, true)$
  result $\leftarrow$ T($main$,$\Gamma$)
```

Figure 18. Transformation of an ASTD specification into BPEL

The function T^{BPEL} initializes the context Γ and starts the transformation algorithm from the main ASTD (see Figure 18). Although the following transformation rules are presented for most ASTD types, the transformation as a whole accounts for the particularities of the patterns presented in the section describing ASTD notation. For concision, instead of using the XML tags for BPEL elements, the following functions use a programming language and prefixed style to construct the code.

```
T($\langle *, b \rangle$,$\Gamma$):
  result $\leftarrow$ **BPEL.RepeatUntil**($false$,T($b$,$\Gamma$))
```

Figure 19. Transformation of an ASTD Kleene closure into BPEL

In Figure 19, $\langle *, b \rangle$ is a Kleene closure, where b is the compound ASTD. The corresponding BPEL code is an infinite loop (**repeatUntil** with its condition set to `false`) over the corresponding BPEL code for the ASTD b.

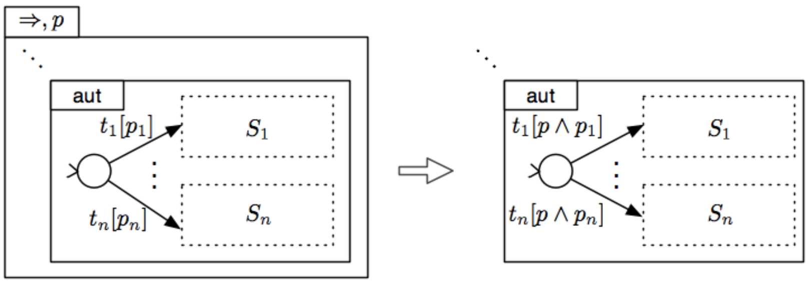


Figure 21. Moving the guard's predicate to the transitions

```
T($\langle \Rightarrow, p, b \rangle$,$\Gamma$):
  $\Gamma' \leftarrow \Gamma \rhd_g \Gamma_g \wedge p$
  result $\leftarrow$ T($b$,$\Gamma'$)
```

Figure 20. Transforming an ASTD guard

To transform an ASTD guard into BPEL, the guard's predicate has to "move" from the ASTD to the first possible transitions of the compound ASTD (see Figure 21). To facilitate this, the context Γ holds a field Γ_g that retains the predicate p in Figure 20 until the first possible transitions are reached.

```
T(⟨|[]|,{t},l,r⟩,Γ):
 $Γ'_l$ ← Γ $⊳_{corr}$ (art_var(l),true)
 $Γ'_r$ ← Γ' $⊳_{corr}$ (art_var(r),true)
 result ← BPEL{.Scope(
[.Variable(whVarEvtHdlt,boolean)].$T_{var}$(t),
[$T_{corr}$(art_var(l)),$T_{corr}$(art_var(r))],
  [$T_{evtHdlr}$(t,l,r,Γ)],
  .Sequence([
   .Assign(whVarEvtHdlt,true),
   .Flow([T(l,$Γ'_l$),T(r,$Γ'_r$)])
   ])
  )}
```

Figure 22. Transformation of a parameterized synchronization into BPEL

```
$T_{evtHdlr}$(t,l,r,Γ):
 result ← BPEL{.OnEvent(
  authDect,
  inAutDecVart,
  .Scope(
   [],[],[],
   .Sequence([
.Assign(outAutDecVart,exp_syn(t,Γ)),
    .Reply(authDect,outAutDecVart),
    .If(
     outAutDecVart.access=granted,
     .Sequence([
.Receive(servExect,inServExect),
.Assign(whVart,inServExect.exec)
     ])
    )
   ])
  )
 )}
```

Figure 23. Transformation of a synchronized event into an event Handler

When dealing with the synchronized event of a parameterized synchronization ASTD, the algorithm creates a BPEL event handler that processes the authorization decision request related to the synchronized event. The event handler runs in parallel with the BPEL code corresponding to the compound ASTD l and r of the parameterized synchronization ASTD (see Figure 22 and Figure 23).

```
T(⟨aut,Σ,N,ν,δ,F,$q_0$⟩,Γ):
 result ← $T_{aut}$($q_0$,⟨aut,Σ,N,ν,δ,F,$q_0$⟩,Γ)
```

Figure 24. Transformation of an automaton: initial processing

The transformation of an ASTD automaton begins at its initial state, as shown in Figure 24. Three cases must be distinguished.

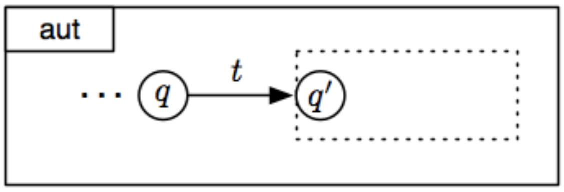


Figure 25. Transformation of an automaton: first case

```
$T_{aut}$(q,automaton,Γ):
 Γ' ← Γ $⊳_g$ true
 forall x | (x,true) ∈ $Γ_{corr}$
  Γ' ← Γ' $⊳_{corr}$ (x,false)
 result ← BPEL.Sequence([
  $T_t$(t,Γ),
  $T_{aut}$(q',automaton,Γ')
 ])
```

Figure 26. BPEL code for the first case

The first case considers a state that has exactly one outgoing transition as may happen in an obligation or SoD pattern instance (see Figure 25). The corresponding BPEL code puts in sequence the BPEL code corresponding to t and the BPEL code yielded by the transformation from the state q', as shown in Figure 26.

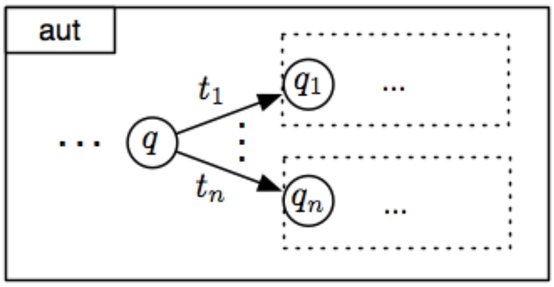


Figure 27. Transformation of an automaton: second case

```
$T_{aut}$(q,automaton,$\Gamma$):
 $\Gamma'$ ← $\Gamma \rhd_g$ true
 forall x | (x,true) ∈ $\Gamma_{corr}$
  $\Gamma'$ ← $\Gamma' \rhd_{corr}$ (x,false)
 result ← BPEL{
  .RepeatUntil(
   false,
   .Pick([
    $T_{onMsg}$($t_1$,$\Gamma$,$T_{aut}$($q_1$,automaton,$\Gamma'$)),
    …
    $T_{onMsg}$($t_n$,$\Gamma$,$T_{aut}$($q_n$,automaton,$\Gamma'$))
   ])
  )
 }
```

Figure 28. BPEL code for the second case

In the second case, the state of interest q has more than one outgoing transition, as illustrated by Figure 27. This case is encountered in permission and prohibition pattern instances. The algorithm in Figure 28 creates a BPEL code that resembles the BPEL code layouts in Figure 15. The last case considers a state q that has no outgoing transitions. For this case, the algorithm creates an **empty** BPEL element (see Figure 29 and Figure 30).

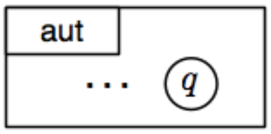


Figure 29. Transformation of an automaton: third case

```
$T_{aut}$(q,automaton,$\Gamma$):
 result ← BPEL.Empty
```

Figure 30. BPEL code for the third case

The first and second cases use the functions T_t and T_{onMsg}, which create the BPEL elements in the case of a simple transition and that of a transition that is part of a **pick** activity. These functions are explained in Figure 31 and Figure 32. In both figures, the helper function exp computes the expression that yields the authorization decision by comparing values from the specification to values received through the authorization decision request.

```
$T_t(t,\Gamma)$:
 if t is synchronized then
  result ← BPEL{.While(
   whVarEvtHdlt,
   .Empty
  )}
  $C_t \leftarrow C_t \rhd exp(t,\Gamma)$
 else
  result ← BPEL{.Scope(
   $T_{var}(t)$, //variables
   [],     //Correlations
   [],     //Event handlers
   .While(
    whVart,
    .Sequence([
     .Receive(authDect,inAutDecVart),
     .Assign(outAutDecVart,exp(t,Γ)),
     .Reply(authDect,outAutDecVart),
     .If(
      outAutDecVart.access=granted,
      .Sequence([
       .Receive(servExect,inSrvExt),
       .Assign(whVart,inSrvExt.exec)
      ])
     )
    ])
   )
  )}
```

Figure 31. Transition in an obligation/SoD pattern instance

```
$T_{onMsg}(t,\Gamma,B)$:
 if t is synchronized then
  // B should be empty
  result ← ""
  $C_t \leftarrow C_t \rhd exp(t,\Gamma)$
 else
  result ← BPEL{.OnMessage(
   (authDect,inAutDecVart),
   [], //Correlations
   .Sequence([
    .Assign(outAutDecVart,exp(t,Γ)),
    .Reply(authDect,outAutDecVart),
    .If(
     outAutDecVart.access=granted,
     .Sequence([
      .Receive(servExect,inSrvExt),
      .Assign(whVart,inSrvExt.exec)
      .If(inSrvExt.exec,B)
     ])
    )
   ])
  )}
```

Figure 32. Transition in a permission/ prohibition pattern instance

Generating the WSDL Interface

A WSDL document for the PDP interface, which includes messages, operations, and port types, is generated from the event list of the ASTD specification according to the algorithm given in Figure 33. The first two message types are independent of events. The message type `authorizationDecisionResponse` is used when the PDP transmits an access decision—which can be granted or denied—to the PEP (point 3 in Figure 14). The message type `serviceExecutedRequest` is sent by the PEP to the PDP to update its state with respect to on whether the IS has or has not executed the requested service.

```
$T^{WSDL}_{spec}$ (spec):
result ←
<definitions name="SecurityFilterProcessInterface">
 <message name="authorizationDecisionResponse">
  <part name="access" type="AccessType"/> <!-- granted or denied -->
 </message>
 <message name="serviceExecutedRequest"/>
  <part name="exec" type="boolean"/>
 </message>
  $T^{WSDL}_{msg}$ ($\langle t^1,\langle p^1_1, T^1_1\rangle,\dots,\langle p^1_{n1}, T^1_{n1}\rangle\rangle$)
  …
  $T^{WSDL}_{msg}$ ($\langle t^r,\langle p^1_r, T^1_r\rangle,\dots,\langle p^r_{nr}, T^r_{nr}\rangle\rangle$)
 <portType name="PDPPortType">
  $T^{WSDL}_{oper}$ ($\langle t^1,\langle p^1, T^1_1\rangle,\dots,\langle p^1_{n1}, T^1_{n1}\rangle\rangle$)
  …
  $T^{WSDL}_{oper}$ ($\langle t^r,\langle p^1_r, T^1_r\rangle,\dots,\langle p^r_{nr}, T^r_{nr}\rangle\rangle$)
 </portType>
 <partnerLinkType name="PDPPartnerLink">
  <role name="securityFilter" portType="PDPPortType"/>
 </partnerLinkType>
</definitions>
```

Figure 33. Transformation of an ASTD specification into a WSDL document

WSDL code snippets are inserted in appropriate places in this document for each event in the list with the appropriate calls to the functions T^{WSDL}_{msg} and T^{WSDL}_{oper}. The former is defined in Figure 34 and produces the message definition used when the PEP requests authorization to execute the service t (point 2 in Figure 14). This message definition contains not only the event t's parameters but also access-control parameters (user u and role r). The latter, defined in Figure 35, produces operation definitions for authorization decision requests/responses as well as the operation used to update the PDP if the IS successfully executed the event t.

```
$T^{WSDL}_{msg}$ ($\langle t,\langle p_1, T_1\rangle,\dots,\langle p_n, T_n\rangle\rangle$):
result ←
<message
name="authorizationDecisiontRequest"/>
  <part name="u" type="UserType"/>
  <part name="r" type="RoleType"/>
  <part name="p1" type="T1"/>
  …
  <part name="pn" type="Tn"/>
</message>
```

Figure 34. Message definition from event signature

```
$T^{WSDL}_{oper}$ ($\langle t,\langle p_1, T_1\rangle,\dots,\langle p_n, T_n\rangle\rangle$):
result ←
<operation name="authDect">
 <input name="inputAuthDect"
message="authorizationDecisiontRequest"/
>
  <output name="outputAuthDect"
message="authorizationDecisionResponse"/
>
</operation>
<operation name="servExect">
 <input name="inputServExect"
  message="serviceExecutedRequest"/>
</operation>
```

Figure 35. Operation definitions from event signature

XSD Types

In ASTD quantified operators, a variable appears explicitly and its value must belong to a predefined set of values, so XSD types must also be created from the ASTD specification. This involves joining a few known types (`AccessType`, `UserType` and `RoleType`) with types gathered from the quantified ASTD (choice and synchronization) of the specification. All other ASTD do not create more types. The function T^{XSD}_{spec} (Figure 36) creates the XSD schema and uses the function T^{XSD} defined recursively on the different ASTD types to create the types required by the BPEL code.

```
$T^{XSD}_{spec}$ (spec):
result ← <schema>
 <simpleType name="AccessType">
  <restriction base="string">
   <enumeration value="granted">
   <enumeration value="denied">
  </restriction>
 </simpleType>
 <simpleType name="UserType" …/>
 <simpleType name="RoleType" …/>
 $T^{XSD}$(main,"")
</schema>
```

Figure 36. Creating the XSD schema for the BPEL process

In ASTD quantified operators, a variable appears explicitly and its value must belong to a predefined set of values, which can be expressed in two forms. The first form is an enumeration, $T=\{x_1, \ldots, x_n\}$. The resulting XSD code is a simple type enumerating all the values in T, as shown in Figure 37. The base type (`integer`, `float`, or `string`) of the simple type is determined by the helper function `base_name` based on the values included in the original set. The second form is an interval of values, $T=[x_l, x_u]$. The resulting XSD code is still a simple type with a range defined from the lower and upper bounds of the interval (see Figure 38). The base type, however, is necessarily integer, since it is the only range base type presently supported. All XSD code snippets are inserted in the same file, which is then imported into the main BPEL process file.

```
T_Type(x,{x_1,…,x_n},xsd):
 result ← xsd +
 <simpleType name="Type_x">
  <restriction
     base="base_name({x_1,…,x_n})">
   <enumeration value="x_1"/>
   …
   <enumeration value="x_n"/>
  </restriction>
 </simpleType>
```

Figure 37. XSD code for an enumeration type

```
T_Type(x,[x_l,x_u],xsd):
 result ← xsd +
 <simpleType name="Type_x">
  <restriction
     base="base_name([x_l,x_u])">
   <minInclusive value="x_l"/>
   <maxInclusive value="x_u"/>
  </restriction>
 </simpleType>
```

Figure 38. XSD code for an interval type

Function T^{XSD} is defined for all ASTD types and relates functions T_{Type}, in Figure 37 and Figure 38, to the main XSD-types function T^{XSD}_{spec} in Figure 36. In Figure 39 and Figure 40, T^{XSD} is given for ASTD quantified choice and quantified synchronization, respectively.

```
T^XSD(⟨|x:T,b⟩,xsd):
 result ← T_Type(x,T,xsd)
```

Figure 39. XSD type for a quantified choice ASTD

```
T^XSD(⟨|[]|x:T,b⟩,xsd):
 result ← T_Type(x,T,xsd)
```

Figure 40. XSD type for a quantified synchronization ASTD

Implementation of Transformations with ATL

We use the principles of model-driven engineering in our projects. Therefore, access-control policy specifications are abstract models; BPEL processes, WSDL interfaces, and XSD-type definitions are concrete models. We use the ATLAS Transformation Language (ATL) (The Eclipse Foundation) to implement transformation from one model into another, as described above, because its framework is well-suited for the conversion of models written in formal languages described with metamodels. The ATL framework is built on the Eclipse platform and provides both a language to express transformation rules and a toolkit to execute them. Since, the metamodels for BPEL and WSDL already exist as XSD models, only a metamodel has been defined for the ASTD notation.

In Figure 41, a model M_1 is transformed into a model M_2 by providing a transformation model (i.e., a set of transformation rules) MM_12MM_2 which maps elements of M_1's metamodel MM_1 to elements of M_2's metamodel MM_2. The metamodels MM_1 and MM_2 as well as ATL are instances of the Ecore metamodel.

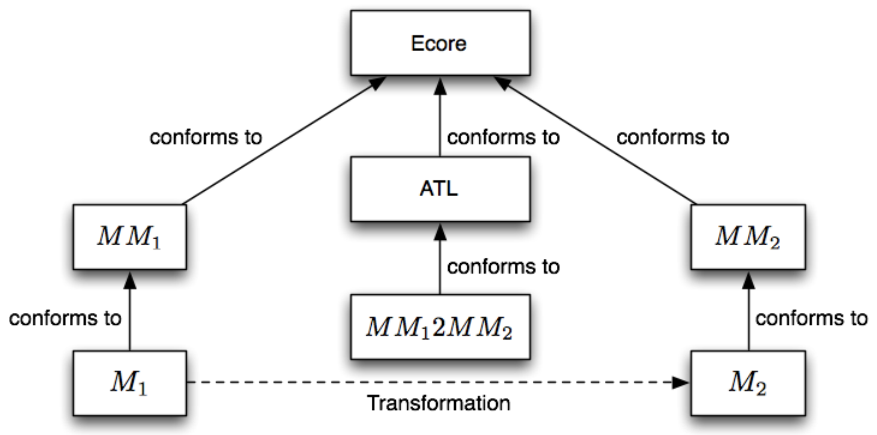


Figure 41. Transforming a model M_1 into a model M_2 using ATL

Related Work

The ORKA (ORKA Consortium) project includes a practical approach to specifying, developing, and deploying access-control policies. In the first phase of the ORKA method, UML and OCL are used to define RBAC-like access-control policies and the associated constraints, respectively (see (INCITS, 2004) for ore information on the role-based access control standard). The constraints express usual notions like SoD and delegation/ revocation. The authors of the ORKA project consider that the UML class diagram of access-control policies and OCL constraints used in conjunction with the USE tool are well-suited for validating security rules (via model checking) against different possible RBAC configurations. Inconsistency or lack of completeness can then be detected. In the second optional phase of the ORKA method, RBAC policies are expressed in CASL (CoFI, 2008) with linear temporal logic formulas. The proofs are performed with the Isabelle theorem prover (Paulson, Nipkow & Wenzel). The latter phase is more comprehensive than the former, since it involves formal proofs. Therefore, this approach was an attempt to fill the gap between practical design for the enforcement of RBAC access-control policies and the use of formal methods to verify them. Even if ORKA, SELKIS, and EB^{3}SEC projects share similar goals, SELKIS and EB^{3}SEC go one step further. Contrary to OCL, ASTD notation has powerful constructs to take into consideration the history of activities, as required, to deal reasonably well with constraints like SoD. Furthermore, since ASTD can be automatically translated into BPEL processes under some assumptions, the implementation approach adopted in our projects seems more appropriate in situations in which such constraints are frequently used. In addition, since ASTD is the only notation used by designers to specify and verify security policies, there is no need to rewrite constraints in another language if formal proof is required, thus avoiding possible errors. Another important difference is at the implementation level. The integration of our enforcement framework into existing applications is a matter of configuring the middleware to route messages from the client to the targeted services through corresponding handlers for the PEP to work correctly. A change in service interface may require modifications in the access-control policy but no modifications at all in the PEP. In the specific PEP mentioned in (Sohr, Mustafa, Bao & Ahn, 2008), such a change would require an update (albeit a small one) to reflect a new interface version.

ASTD notation is not the only one that has been adapted to specify RBAC-like access-control policies. Access-right constructs have recently been added to CaSPiS (Calculus of Services with Pipelines and Sessions) notation (Kolundžija, 2009), which is a calculus to specify WS by explicitly defining sessions (conversations between clients and servers) and properties (like graceful termination) (Boreale, Bruni, De Nicola & Loreti, 2008). In its original version, CaSPiS provides a denotational semantics, which has been extended to accommodate access rights. To the best of our knowledge, no further work has been done to exploit this notation in a practical framework.

In our projects, transformations are mainly used to obtain implementation code from a high-level specification. Transformations can also be used in the modeling phase to derive secrecy models from a base (UML) model by iteratively applying property preserving transformation operations as proposed by Hassan et al. (2010). This work is still in its early stages. The main drawback seems related to the expressiveness of UML models, which are limited to the RBAC level of Figure 13.

Basin et al. (2009) used separation of duty algebra (SoDA), an algebra for SoD developed by Li and Wang (2006), to implement access-control policies in a framework in which workflows are modeled with CSP. This methodology cannot be used to express other type of constraints like obligation, as is possible in our projects. Other work has been proposed to implement access control on WS. Bertino et al. (2006) developed a framework to apply RBAC policies to WS with XACML encoding. Since RBAC itself is not tailored for SoD constraints, their framework used Business Process Constraint Language (BPCL) to express both SoD rules and a broader range of constraints. Yao et al. (2001) presented an architecture for access control built around services with support for formal RBAC-like policies. The history of actions is, however, not take into account in this framework. Jajodia et al. (2001) proposed a language that supports multiple access-control policies for a single system with a focus on conflict resolution. They provided the notion of history through a history table. All these frameworks have the same drawback: when they do not support constraints like SoD, they use different notations or mechanisms to overcome the limitation. We use ASTD as a unified, intuitive, and powerful notation to express permissions as well as various constraints, including ordering constraints and SoD, in particular.

Other work has been done on BPEL and formal notations. Wong and Gibbons (2007) described an approach for designing workflows using CSP. Their methodology is based on the specification of some control flow and state-based patterns originally proposed by van der Aalst. He defined a formal model of workflows in (van der Aalst, 1998) using Petri nets, later enhanced by Massuthe et al. (2005). Both notations have each their disadvantages with respect to access control and verification. On the one hand, CSP does not support state variables and is not well-suited for liveness properties. On the other, Petri nets do not support quantification, which is an important feature when dealing with IS.

Conclusion

The aim of the method presented in this paper is to deploy access-control policies in a PDP automatically. Security managers are able to specify such policies at the process level using rigorous notation. The policies are enacted in a BPEL engine as part of an access right enforcement framework. In our future work, PEP and PDP will be integrated into a larger framework in which additional functionalities (e.g., policy edition for the three levels in Figure 13) will be provided. The drawback of the approach presented herein is linked to the transformation method. Indeed, the BPEL process derived by applying transformation rules reproduces the event flow specified by an ASTD. When the policy is updated, the BPEL process and its current execution state have to be recreated from scratch. One way to avoid this is to record ASTD as XML documents and exploit a BPEL engine to interpret these documents. Specifically, an ASTD specification would be transformed into an XML variable in the BPEL process, as would the current state of the ASTD. Since BPEL is not suitable for data manipulation (the **assign** functionality is rather rudimentary), an interpreter would require a BPEL engine that provides extension points to deal with adequate XML data manipulation and complex computation. For instance, Oracle BPEL Process Manager and GlassFish ESB v2.1 support data manipulation through Java and Javascript code, respectively. Finally, experiments should be conducted to evaluate and compare both solutions for applications of realistic size.

Acknowledgement

The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the French National Research Agency (ANR) as part of the SELKIS project (ANR-08-SEGI-018).

References

Abrial, J.-R. (2010). Modeling in Event-B. Zürich, Swiss: Cambridge University Press.

Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., & Voisin, L. (2010). Rodin: An open toolset for modelling and reasoning in Event-B. Software Tools for Technology Transfer, 12(6), 447-466.

Aït-Sadoune, I., & Aït-Ameur, Y. (2010). Stepwise design of BPEL Web services compositions: An Event-B refinement based approach. In R. Lee, O. Ormandjieva, A. Abran, & C. Constantinides (Eds.), Software Engineering Research, Management and Applications: Vol. 296. Studies in Computational Intelligence (pp. 51-68). Springer, Berlin Heidelberg.

Basin, D. A., Burri, S. J., & Karjoth, G. (2009). Dynamic enforcement of abstract separation of duty constraints. In M. Backes, & P. Ning (Eds.), Computer Security - ESORICS 2009: Vol. 5789. Lectures Notes in Computer Science (pp. 250-267). Springer, Berlin Heidelberg.

Bertino, E., Crampton, J., & Paci, F. (2006). Access control and authorization constraints for WS-BPEL. Proceedings of the IEEE International Conference on Web Services (pp. 275-284). Chicago, IL: IEEE Computer Society.

Bolognesi, T., & Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems, 14(1), 25-59.

Boreale, M., Bruni, R., De Nicola, R., & Loreti, M. (2008). Sessions and pipelines for structured service programming. In G. Barthe, & F. de Boer (Eds.), Formal Methods for Open Object-Based Distributed Systems: Vol. 5051. Lecture Notes in Computer Science (pp. 19-38). Springer, Berlin Heidelberg.

CoFI (2008-12-02). CASL - CoFI. Retrieved 2011-29-01 from CASL Web Site: http://www.informatik.uni-bremen.de/cofi/wiki/index.php/CASL

Embe Jiague, M., Frappier, M., Gervais, F., Konopacki, P., Laleau, R., Milhau, J., & St-Denis, R. (2010). Model-driven engineering of functional security policies. In J. Filipe, & J. Cordeiro (Eds.), International Conference on Enterprise Information Systems: Vol. 3. Information Systems Analysis and Specification (pp. 374-379). INSTICC Press.

Frappier, M., Gervais, F., Laleau, R., & Fraikin, B. (2008). Algebraic state transition diagrams (Tech. Rep. No. 24). Sherbrooke, Québec : Université de Sherbrooke, Département d'informatique.

Frappier, M., Gervais, F., Laleau, R., Fraikin, B., & St-Denis, R. (2008). Extending statecharts with process algebra operators. Innovations in Systems and Software Engineering, 4(3), 285-292.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3), 231-274.

Hassan, W., Slimani, N., Adi, K., & Logrippo, L. (2010). Secrecy UML method for model transformations. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, & S. Reeves (Eds.), Abstract State Machines, Alloy, B and Z: Vol.

5977. Lecture Notes in Computer Science (pp. 16-21). Springer, Berlin Heidelberg.

Hoare, C. A. (1978). Communicating sequential processes. Communications of the ACM, 21(8), 666-677.

INCITS (2004). Role base access control. INCITS. American National Standard for Information Technology.

Jajodia, S., Samarati, P., Sapino, M. L., & Subrahmanian, V. S. (2001). Flexible support for multiple access control policies. ACM Transactions on Database Systems, 26(2), 214-260.

Kolundžija, M. (2009). Security types for sessions and pipelines. In R. Bruni, & K. Wolf (Eds.), Web Services and Formal Methods: Vol. 5387. Lecture Notes in Computer Science (pp. 175-190). Springer, Berlin Heidelberg.

Konopacki, P., Frappier, M., & Laleau, R. (2010a). Expressing access control policies with an event-based approach (Tech. Rep. No. TR-LACL-2010-6). Créteil, France : Université Paris 12, Laboratoire d'Algorithmique, Complexité et Logique.

Konopacki, P., Frappier, M., & Laleau, R. (2010b). Modélisation de politiques de sécurité à l'aide d'une algèbre de processus. RSTI - Ingénierie des systèmes d'information, 15(3), 113-136.

Li, N., & Wang, Q. (2006). Beyond separation of duty: An algebra for specifying high-level security policies. 13th ACM Conference on Computer and Communications Security (pp. 356-369). Alexandria: ACM.

Massuthe, P., Reisig, W., & Schmidt, K. (2005). An operating guideline approach to the SOA. Annals of Mathematics, Computing & Teleinformatics, 1, 35-43.

Milhau, J., Frappier, M., Gervais, F., & Laleau, R. (2010). Systematic translation rules from ASTD to Event-B. In D. Méry, & S. Merz (Eds.), Integrated Formal Methods: Vol. 6396. Lecture Notes in Computer Science (pp. 245-259). Springer, Berlin Heidelberg.

OASIS (2005). eXtensible Access Control Markup Language (XACML) Version 2.0. Organization for the Advancement of Structured Information Standards.

OASIS (2007). Web Services Business Process Execution Language Version 2.0. Standard, Organization for the Advancement of Structured Information Standards.

ORKA Consortium (n.d.). ORKA - Organizational Control Architecture - Overview. Retrieved 2011-29-01 from Project ORKA Web Site: http://www.organisatorische-kontrolle.de/index-en.htm

Paulson, L., Nipkow, T., & Wenzel, M. (n.d.). Isabelle. Retrieved 2011-29-01 from Isabelle Web Site: http://www.cl.cam.ac.uk/research/hvg/Isabelle/

Sohr, K., Mustafa, T., Bao, X., & Ahn, G.-J. (2008). Enforcing role-based access control policies in Web services with UML and OCL. Proceedings of the 4th Annual Computer Security Applications Conference (pp. 257-266). Washington, DC: IEEE Computer Society.

The Eclipse Foundation (n.d.). ATL. Retrieved 2011-29-01 from ATL Web Site: http://www.eclipse.org/atl/

van der Aalst, W. M. (1998). The application of Petri nets to workflow management. The Journal of Circuits, Systems and Computers, 8(1), 21-66.

Wong, P. Y., & Gibbons, J. (2007). A process-algebraic approach to workflow specification and refinement. In M. Lumpe, & W. Vanderperren (Eds.), Software Composition: Vol. 4829. Lecture Notes in Computer Science (pp. 51-65). Springer, Berlin Heidelberg.

Yao, W., Moody, K., & Bacon, J. (2001). A model of OASIS role-based access control and its support for active security. Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (pp. 171-181). Chantilly: ACM.

Troisième partie

ASTD + SecureUML + UML vers B0sec

Chapitre 2

Combinaison d'UML, ASTD et B pour la spécification formelle d'un filtre de contrôle d'accès

Dans ce chapitre, nous nous intéressons à la spécification en B d'un filtre de contrôle d'accès. Ce filtre est composé de plusieurs machines B dont au moins une est issue d'une traduction d'un modèle exprimé avec la notation ASTD. Ainsi ce chapitre combine la machine traduite avec d'autres machines dans le but d'obtenir un modèle formel de filtre de contrôle d'accès qui inclut le modèle fonctionnel du système ainsi que les modèles de la politique de contrôle d'accès. L'objectif est de valider le comportement complet du couple filtre + système.

La combinaison de méthodes formelles et semi-formelles est de plus en plus nécessaire pour produire des spécifications qui peuvent à la fois être comprises et donc validées par le concepteur et l'utilisateur mais aussi suffisamment précises pour être vérifiées via des approches formelles. Cela motive notre approche d'utiliser des paradigmes complémentaires afin de spécifier certains aspects de la sécurité des systèmes d'information. Cet article présente une méthodologie pour spécifier des politiques de contrôle d'accès des systèmes d'information à l'aide de notations graphiques : UML pour le modèle fonctionnel, SecureUML pour le modèle statique de contrôle d'accès et ASTD pour le modèle dynamique de contrôle d'accès. Ces diagrammes sont traduits en un ensemble de machines B. Finalement, nous présentons la spécification formelle d'un filtre de contrôle d'accès qui coordonne les différentes règles de contrôle d'accès et la spécification fonctionnelle des opérations. Le but de ces spécifications B est de vérifier rigoureusement la politique de contrôle d'accès d'un système d'information en utilisant les nombreux outils de la méthode B.

Une première version de cet article a été soumise et acceptée à la quatrième édition du workshop international IEEE *UML and Formal Methods* (UML&FM'2011) ayant eu lieu à Limerick en Irlande le 20 juin 2011. Nous présentons ici la version étendue et acceptée au journal *Innovations in Systems and Software Engineering (ISSE).*

Combining UML, ASTD and B for the Formal Specification of an Access Control Filter

Jérémy Milhau [1,2], Akram Idani[3], Régine Laleau[2], Mohamed-Amine Labiadh[3], Yves Ledru[3], Marc Frappier[1]

1 : GRIL, Université de Sherbrooke, 2500 Boulevard de l'Université, Sherbrooke QC, Canada
{Jeremy.Milhau,Marc.Frappier}@usherbrooke.ca

2 : LACL, Université Paris-Est, IUT Sénart Fontainebleau, Département Informatique
Route Forestière Hurtault, 77300 Fontainebleau, France
{Frederic.Gervais,Laleau}@u-pec.fr

3 : UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble2/CNRS
Laboratoire d'Informatique de Grenoble UMR 5217, 38041 Grenoble, France
{Yves.Ledru,Akram.Idani,Mohamed-Amine.Labiadh}@imag.fr

Abstract Combination of formal and semi-formal methods is more and more required to produce specifications that can be, on the one hand, understood and thus validated by both designers and users and, on the other hand, precise enough to be verified by formal methods. This motivates our aim to use these complementary paradigms in order to deal with security aspects of information systems. This paper presents a methodology to specify access control policies starting with a set of graphical diagrams: UML for the functional model, SecureUML for static access control and ASTD for dynamic access control. These diagrams are then translated into a set of B machines. Finally, we present the formal specification of an access control filter that coordinates the different kinds of access control rules and the specification of functional operations. The goal of such B specifications is to rigorously check the access control policy of an information system taking advantage of tools from the B method.

Keywords Access control policy, B, SecureUML, ASTD

2.1 Introduction

Our aim is the formal specification of information systems (IS) access control policies. Roughly speaking, an IS helps an organization to collect and manipulate all its relevant data. Access control is part of security issues that can grant or deny the execution of actions depending on a policy. An access control policy defines, for an authenticated user, which actions he is allowed or forbidden to execute depending on several criteria such as role, organization, etc. It is the combination of atomic rules. Depending on the kind of rules an access control designer wants to express, several languages and notations can be used. In an access control policy specification, static and dynamic rules may be required in order to express all access control requirements. In our work, we consider static rules independently of the functional behavior of the system. Permissions, prohibitions and static Separation of Duty (SoD) are static constraints. A static SoD constraint means that if a user is assigned to one role, he is prohibited from being a member of a second role [10]. Dynamic constraints require to take into account the history of the system, that is the set of actions already performed on a system, which is represented by the system state and its evolutions. For instance, obligations and dynamic SoD are dynamic constraints. With dynamic SoD, users may be authorized for roles that may conflict, but limitations are imposed, based on the history of actions performed and roles taken by the user.

In our approach, we have chosen to use SecureUML [25] to express static rules and the ASTD notation [13] for dynamic rules. These notations are presented in Section 2.3. These graphical specifications are then translated into B machines [1] using translation rules described in Section 2.4. Similarly, as we need also to specify the functional model of the IS since access control rules can refer to elements of this model, its UML specification is translated into B. The next step consists in specifying the access control filter that coordinates the different kinds of access control rules and the functional model. The B specification of this filter is presented in Section 2.5. We start in the next section by describing the context of the work.

2.2 Context

2.2.1 The Selkis Project

The Selkis project [1] funded by the French national research agency (ANR) aims to define a development strategy for secure healthcare network IS from requirements engineering to implementation. The results of this project can be applied to any type of secure IS, but the medical field was chosen because of the complexity and diversity of security requirements. An approach based on formal methods was chosen in order to build implementations correct by design and to perform proofs and model checking over rules that check properties of the specification.

The approach adopted in the Selkis project advocates a separation between the access control policy and the functional model at the requirements, specification and implementation levels. An implementation of an access control filter is produced. It intercepts actions before they are executed. If the action and other parameters match access control rules, the action can be executed by the IS. In the other case, the execution is denied.

2.2.2 Illustrative Example

In order to illustrate our approach, we consider an example from a medical information system. The structural representation of this example is modeled by the UML class diagram of Fig. 2.1. This diagram manages medical information about patients in a hospital. Class MedicalRecord stores medical information about a given patient, such as his medications, using the attribute *data*. Every patient has at most one medical record which is not depending of hospitals. A doctor can practice in at most one hospital and he can leave and join a hospital using methods *joinHospital* and *leaveHospital*. The information system imposes the following constraint:

C1. A patient cannot leave a hospital if his medical record is not validated,

The access control policy associated to our example includes the following rules:

1. http://lacl.fr/Selkis/

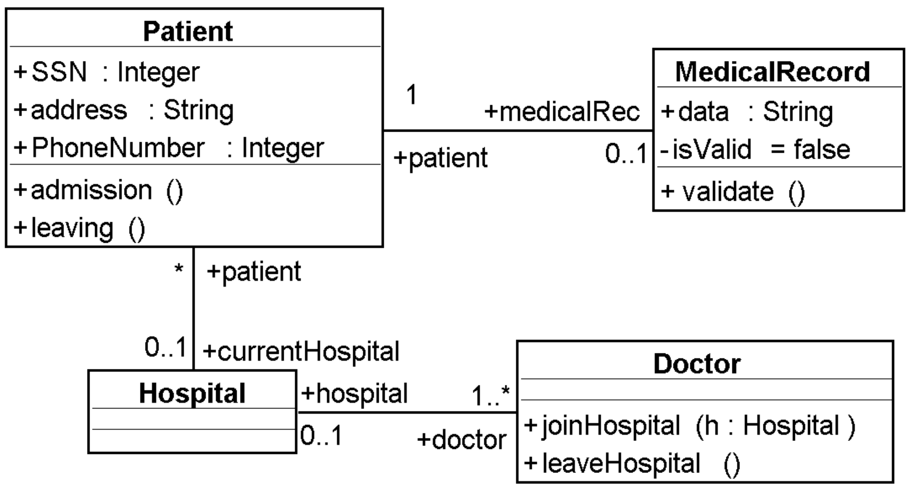


Figure 2.1: Functional model

- R1 Doctors can read both public and private attributes of a medical record but they can only modify public attributes (*i.e.* data),
- R2 Doctors can only validate medical records using the *validate* method,
- R3 Modification and validation of a medical record of a given patient, can only be done by a doctor who does belong to the same hospital as the patient,
- R4 If a patient has left the hospital, only doctors belonging to the hospital during the patient's stay will keep read access to his medical record.
- R5 Any modification of a patient's medical record must be eventually validated. Several modifications can be validated by a single validation.

Other rules exist in order to define the permissions to execute actions of classes *Patient* and *Doctor* but they are not presented in this paper for the sake of concision.

2.3 Graphical Models for Access Control

In our methodology we propose to use SecureUML [25] to model static access control, and ASTD [13] diagrams for dynamic access control. Static access control is based on RBAC (Role-Based Access Control) [34], which describes authorizations granted to users on resources. Dynamic access control rules can refer to previous states of the IS.

2.3.1 SecureUML

SecureUML [25] is a graphical modeling language designed to integrate information relevant to access control into application models defined with the Unified Modeling Language (UML). It extends a functional UML model using concepts of role-based access control models (RBAC) in order to model roles and their permissions. This is the main reason why we have chosen SecureUML rather than UMLSec [19]. In SecureUML, users are grouped into roles and may play several roles with respect to the secure system. Fig. 2.2 uses concepts of SecureUML to model rules R1 and R2. It shows how medical records are secured when they are accessed by doctors.

This SecureUML specification indicates that users with role *Doctor* can read private and public attributes (denoted by ≪`EntityAction`≫ `privateRead`), modify public attributes (≪`EntityAction`≫

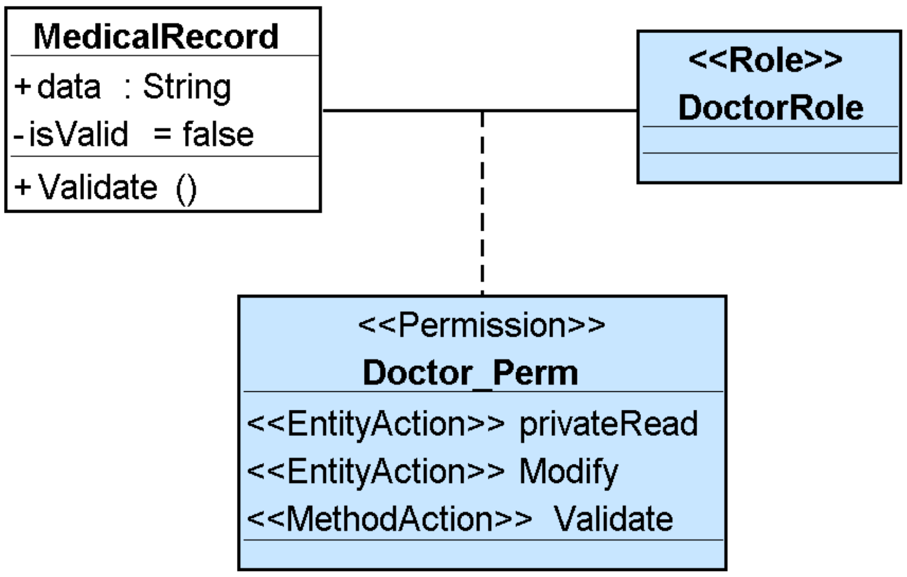


Figure 2.2: Access control rules for medical records

`Modify`). This part of the SecureUML specification models rule R1. Rule R2 is modeled by the permission that allows doctors to execute the method of the class *MedicalRecord*: *Validate* (≪`MethodAction`≫ `Validate`).

Rule R3 refers to an authorization constraint associated to the permission to access a medical record and which links the security and the functional parts of this model. It requires (i) to navigate through the functional model to retrieve the patient associated to the medical record, and his/her current hospital, (ii) to retrieve the doctor corresponding to the user asking to access the medical record and retrieve his/her associated hospital, (iii) to compare these two hospitals. We can add this constraint by annotating the graphical SecureUML model. However, in order to keep Fig. 2.2 readable, rule R3 is described using a B predicate in Section 2.4.3.

It would be more difficult to express rule R4 using SecureUML, but not impossible. Rule R4 is of dynamic nature because it is based on information about past states of the system. Thus, in order to express it in SecureUML, it would be necessary to add few artificial variables to the functional model to store this kind of information. However, for larger functional models which deal with real information systems it becomes error prone to do so because these variables will be less manageable. This kind of variables is not needed when using the ASTD notation as it offers features streamlining the specification of dynamic aspects. We think that using a graphical notation explicitly modeling states and transitions between states is more intuitive than coding manually state variables that may introduce errors in the specification. This approach of coding states into variables is generally used in SecureUML, since it does not provide operators to specify ordering constraints between actions.

2.3.2 The ASTD Notation

The ASTD notation is a graphical representation having a formal semantics. It was created to specify systems, in particular IS. ASTD was introduced as an extension of Harel's Statecharts [16] and is based on operators from EB^3 [14] (a process algebra dedicated to IS specifications). Readers are invited to consult a formal and mathematical description of the ASTD notation in [13].

In our IS hospital example, access control rule R4 is a dynamic rule since it refers to several actions and defines ordering constraints upon them. Hence this rule is modeled using the ASTD notation as described in Fig. 2.3. The ASTD uses a notation separating actions of the IS and its parameters from security parameters such as the user and his/her role in the IS. It is denoted $< \overrightarrow{s}, a(\overrightarrow{p}) >$ where $\overrightarrow{s}$ is the list of

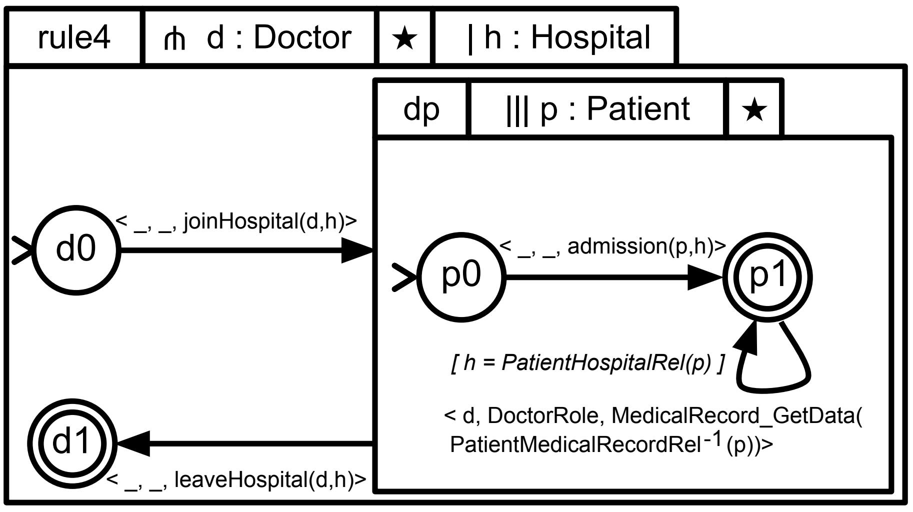

rule4
⫛ d : Doctor
★
| h : Hospital
dp
||| p : Patient
★
d0
< _, _, joinHospital(d,h)>
p0
< _, _, admission(p,h)>
p1
[h = PatientHospitalRel(p)]
< d, DoctorRole, MedicalRecord_GetData(PatientMedicalRecordRel⁻¹ (p))>
d1
< _, _, leaveHospital(d,h)>

Figure 2.3: ASTD model of rule R4: If a patient has left the hospital, only doctors belonging to the hospital during the patient's stay will keep read access to his medical record

access control parameters (user and role in our example), a is the action the user wants the IS to execute and $\overrightarrow{p}$ are the functional parameters of the action. Since in our example the combination of user and role is checked in the SecureUML model, there is no need to check it again in the ASTD model. Some security parameters are wildcards (denoted _), meaning that the specification accepts any values for these parameters. However, if specific constraints upon security parameters are required, they can be specified in the ASTD for instance in the action MedicalRecord_GetData.

The first operator denoted $⫛\ d : Doctor$ is a quantified weak synchronization over all doctors of the system. There are as many instances of the quantified ASTD as the number of instances of class *Doctor*. When an action is received, all the instances of the ASTD that can execute it do so at the same time. In other words, if there are instances of the ASTD that can synchronize, they have to do so. The quantified choice operator $|\ h : Hospital$ means that a single instance h of class *Hospital* is associated to the instance d of *Doctor*. This link is created by action JoinHospital(d, h) when a doctor is assigned to a hospital. The link between these instances is removed when the doctor leaves the hospital with the action LeaveHospital(d, h). After leaving a hospital, a doctor can join another one, starting a new link between d and h. This iteration is possible thanks to the Kleene closure operator (denoted by $\star$), meaning that the sub-ASTD can be iterated as many times as needed. Finally, ASTD named dp describes the ordering constraint of action admission(p, h) and action :

$$< d, DoctorRole, \text{MedicalRecord\_GetData}(i) >$$
$$\text{with } i = PatientMedicalRecordRel^{-1}(p)$$

for all instances of the *Patient* class (due to operator quantified interleave $|||p : Patient$). This action must be executed by d, a user who has the role *DoctorRole*. It is guarded by the predicate

$$h = PatientHospitalRel(p)$$

in order to check that the hospital of the patient p and h, the hospital where d works, are the same.

Similarly, we can model rule R5 that depicts an obligation. After one or several modifications of a given medical record, the record must be validated. Rule R5 is modeled in Fig. 2.4 using the ASTD notation. This ASTD describes the process for all medical records, and this process can be repeated. This is modeled using $|||\ Instance : MedicalRecord$ and the Kleene closure *. Then we have an automaton describing the ordering of actions *MedicalRecord_SetData* and *MedicalRecord_Validate*. This automaton

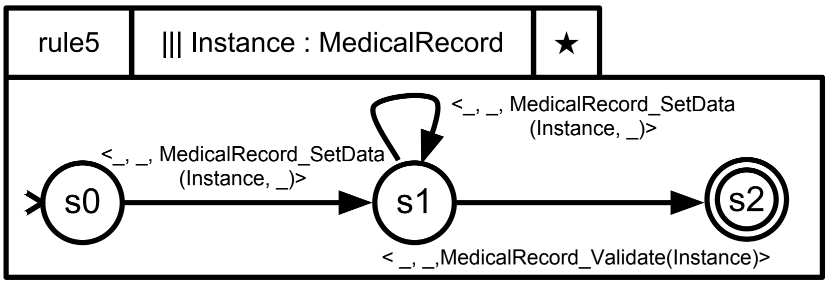

rule5
||| Instance : MedicalRecord
★
<_, _, MedicalRecord_SetData (Instance, _)>
<_, _, MedicalRecord_SetData (Instance, _)>
s0
s1
s2
< _, _,MedicalRecord_Validate(Instance)>

Figure 2.4: ASTD model of rule R5: Any modification of a patient's medical record must be eventually validated. Several modifications can be validated by a single validation

means that one or several *MedicalRecord_SetData* can be executed and are eventually followed by single *MedicalRecord_Validate*.

In OrBAC [6], the notion of context may be useful for some dynamic constraints. Indeed, contexts allow to refine permissions in order to give rights in specific circumstances (*e.g.* emergency situations). They can govern periods of validity of privileges, or reduce/extend the access rights inherited from a role. Especially, OrBAC proposes the notion of provisional context [5] which depends on previous actions the user has performed in the system. This assumes that the information system manages a log that stores data about previous activities of users in the system. The OrBAC approach requires then the effective implementation of the system. In our approach, the advantage of ASTD models, compared with OrBAC provisional contexts, is that they address the conceptual phases of a secure IS development process rather than implementation or runtime.

2.4 Translations into B Specifications

The idea of integrating formal and graphical notations (*i.e.* B and UML) has been studied since several years [15], and was commonly motivated by the complementarities of these two types of notation. Indeed, the disadvantages of semi-formal methods can be overcome thanks to contributions of formal methods and vice versa. UML graphical views are synthetic, structural and intuitive. However, their semantics are often described as blurred. Therefore, the construction of systems based on these methods can sometimes lead to ambiguous models. On the other hand, the major strengths of formal methods are precision of the abstract mathematical notations and automatic reasoning.

In our approach, we propose to translate both SecureUML and ASTD specifications into B in order to check the global consistency of access control policies. Moreover we also need the functional model since security rules can require to read functional attributes values. Thus, the translation into B is composed of three steps corresponding to the functional, static access control and dynamic access control models.

2.4.1 The B Method

The B method [1] is a method that supports a large segment of the software development life cycle: specification, refinement and implementation. It ensures, thanks to refinement steps and proofs, that the code matches to the specification. The B method is based on Abstract Machine Notation (AMN) and the use of formally proved refinements. Its mathematical basis is founded on first-order logic, integer arithmetic and set theory. A B specification is structured into machines which contain state variables, invariant properties expressed on the variables and operations specified in the generalized substitution language, which is a generalization of Dijkstras' guarded command language. The refinement mechanism consists in reformulating, by successive steps, the variables and the operations of an abstract machine, so as to finally lead to a module which will constitute a running program. The intermediate steps of reformulation are called refinements and the last one is the implementation.

2.4.2 A Formal Functional Model

The initial functional model (fig. 2.1) is a UML class diagram showing entities and relationships. In order to formally reason on this model, we propose to translate it into the B notation. Translation from UML diagrams into B specifications was addressed by several research works [20, 21, 35]. In order to take advantage of these complementary works we integrated their translation rules in a unified MDE framework [18]. This allows, on the one hand, to combine and adapt their rules, and on the other hand, to extend them in order to take into account translations of UML extensions (*i.e.* SecureUML). The translation of the functional model is strongly inspired by these approaches. We do not use the UML-B [36] framework of RODIN toolset because our objectives are quite different. On the one hand, the RODIN toolset is dedicated to the Event-B language and on the other hand the UML-B [36] approach gives a UML syntax to B which does not cover UML constructs that we need in our approach such as composition or navigation.

Translation principles. The functional model is translated into a unique B machine containing sets, variables and associations derived from classes, attributes, and class relations. As proposed by the existing approaches, class named *MedicalRecord* of Fig. 2.1 leads to an abstract set *MEDICALRECORD* and a variable *MedicalRecord* representing respectively the set of possible instances and the set of existing instances of class *MedicalRecord*. Fig. 2.5 illustrates basic B structures related to class *MedicalRecord*.

Our MDE platform generates basic operations such as constructors, destructors, getters, setters in order to allow state evolution of the functional formal model. This step takes into account some basic structural invariants related to mandatory and/or unique attributes, inheritance, composition, multiplicities... An example of a basic getter which allows to get medical record data is given in Fig. 2.6.

The operation returns a medical record data and information about the success of its execution. This last point is useful for the access control filter detailed later. From a functional point of view, reading medical records data should always succeed.

B specifications resulting from our translation process can be enriched in order to take into account less obvious functional constraints. Let us consider for example constraint C1 which means that if attribute *isValid* of a medical record is *false* then the patient of this medical record must be linked to some hospital. This can be expressed by the following invariant:

$$\boxed{\begin{array}{l}\forall\, pp \cdot (pp \in Patient \wedge \\ MedicalRecord\_\_isValid(PatientMedicalRecordRel^{-1}(pp)) = \mathbf{FALSE} \Rightarrow Pa\text{-} \\ tientHospitalRel[\{pp\}] \neq \varnothing\ )\end{array}}$$

Contrary to operation *MedicalRecord__GetData*, the success of operation *MedicalRecord__SetData* is constrained by C1. In fact, this basic setter modifies attribute *data* of a MedicalRecord and also sets the attribute *isValid* to *false*. Fig. 2.7 presents this B operation and shows how the previous invariant is respected.

Failure of *MedicalRecord__SetData* indicates to the access control filter that someone tried to modify data of a medical record of a patient who is not admitted in any hospital and that this action is forbidden by the functional part of the model.

```
MACHINE
    Functional_Model
SETS
    MEDICALRECORD ;
    THEDATA ;
    ...
ABSTRACT_VARIABLES
    MedicalRecord ,
    MedicalRecord__data ,
    MedicalRecord__isValid ,
    ...
INVARIANT
    MedicalRecord ⊆ MEDICALRECORD ∧
    MedicalRecord__data ∈ MedicalRecord ⇸ THEDATA ∧
    MedicalRecord__isValid ∈ MedicalRecord → BOOL ∧
    PatientMedicalRecordRel ∈ MedicalRecord ↣ Patient
    ...
OPERATION
executed ← MedicalRecord__Validate(Instance) ≙
    PRE
        Instance ∈ MedicalRecord∧
        MedicalRecord__isValid ( Instance ) = FALSE
    THEN
        MedicalRecord__isValid ( Instance ) = TRUE ;
        executed := ok
    END;
```

Figure 2.5: Basic B structures related to medical records

2.4.3 A Formal Static Access Control model

To our knowledge there is no attempt to translate secureUML models into B. Nevertheless, in [33] the authors gave an Event-B specification for OrBAC policies which is mainly dedicated to formally prove the process of deploying a security policy. In our work, we mainly deal with the modeling level of a security policy and its impact on the functional model.

In order to translate the security part of our model, we propose a mapping which leads to structures that represent data types. First, we propose a B formalization of a variant of the SecureUML meta-model. Then, the security model elements are directly injected in this B specification. For example, in the following we give some enumerated sets issued from Fig. 2.2.

```
SETS
    ENTITIES = {MedicalRecord ...};
    ATTRIBUTES = {data ...};
    ACTIONS = {MedicalRecord__SetData ...};
    PERMISSIONS = {Doctor_Perm ...};
    ROLES = {DoctorRole ...};
    ...
```

Invariants of the security formal model define various relations between these elements and also structural constraints imposed by the meta-model. For example, permission assignments and role hierarchy are defined by:

```
$result, executed \leftarrow MedicalRecord\_\_GetData(Instance) \triangleq$
    PRE
        $Instance \in MedicalRecord$
    THEN
        $result := MedicalRecord\_\_data(Instance) \parallel$
        $executed := ok$
    END;
```

Figure 2.6: Operation *MedicalRecord__GetData*

```
$executed \leftarrow MedicalRecord\_\_SetData(Instance, data) \triangleq$
    PRE
        $Instance \in MedicalRecord \wedge$
        $data \in THEDATA$
    THEN
        IF
            $PatientMedicalRecordRel(Instance) \in$
                        $\mathrm{dom}(PatientHospitalRel)$
        THEN
            $MedicalRecord\_\_data(Instance) := data \parallel$
            $MedicalRecord\_\_isValid(Instance) := FALSE \parallel$
            $executed := ok$
        ELSE
            $executed := failure$
        END
    END;
```

Figure 2.7: Operation *MedicalRecord__SetData*

$$\mathrm{PermissionAssignement} \in PERMISSIONS \rightarrow (ROLES \times ENTITIES) \wedge Roles\_Hierarchy \in ROLES \leftrightarrow ROLES \wedge \mathbf{closure1}(Roles\_Hierarchy) \cap \mathbf{id}(ROLES) = \varnothing$$

This means that a permission links at the most one pair (role $\mapsto$ entity), and that *Roles_Hierarchy* has no cycle. Invariants also include RBAC constraints such as the definition of SSD (Static Separation of Duty) which forbids a user to take conflicting roles even in different sessions.

The initialisation clause valuates the various relations according to the SecureUML meta-model instance. This allows to check that the security model respects the structural constraints of the SecureUML meta-model such as the non-circular role hierarchy. A brief overview of the initialisation clause is given below:

```
INITIALISATION
$AttributeOf := \{(data \mapsto MedicalRecord), \ldots\}$
$AttributeKind := \{(data \mapsto public), \ldots\}$
$OperationOf := \{(MedicalRecord\_\_SetData \mapsto MedicalRecord), \ldots\}$
$setterOf := \{(MedicalRecord\_\_SetData \mapsto data), \ldots\}$
$PermissionAssignement :=$
    $\{(Doctor\_Perm \mapsto (DoctorRole \mapsto MedicalRecord)), \ldots\}$
$\mathrm{EntityActions} := \{(Doctor\_Perm \mapsto \{privateRead, modify\}), \ldots\}$
...
```

Operations of the B specification derived from the security model are dedicated to control access to the operational part of the functional formal model. We associate to each functional operation a secured operation in the security model which verifies, based on the initial state, that a user has permission to call the functional operation. For example, operation *secure_MedicalRecord__SetData* presented in Fig. 2.8 is intended to verify accesses to the functional operation *MedicalRecord__SetData* of Fig. 2.7. Secured operations add parameters *user* and *role* corresponding respectively to the user who is trying to invoke the operation and one of his roles ($role \in roleOf(user)$). Predicate "$MedicalRecord\_\_SetData \in isPermitted[\{role\}]$" verifies whether operation *MedicalRecord__setData* is allowed to the connected user using a particular role. Indeed, set *isPermitted* computes, from the initial state, the set of authorized functional operations for each role. For instance, it contains the couple ($DoctorRole \mapsto MedicalRecord\_\_SetData$).

```
answer ← secure_MedicalRecord__SetData(Instance, user, role) ≙
    PRE
        Instance ∈ MedicalRecord ∧
        user ∈ USERS ∧ role ∈ ROLES ∧
    THEN
        IF
            role ∈ roleOf(user) ∧
            MedicalRecord__SetData ∈ isPermitted[{role}] ∧
            (user ∈ Doctor ⇒ HospitalDoctorRel(user) =
            PatientHospitalRel(PatientMedicalRecordRel(Instance)))
        THEN
            answer := granted
        ELSE
            answer := denied
        END
    END
```

Figure 2.8: Operation *secure_MedicalRecord__SetData*

As mentioned in Section 2.3.1, rule R3 refers to a constraint which is added to the SecureUML model using an annotation linked to permission *Doctor_Perm*. It is expressed in the B language as a precondition of modification actions (`≪EntityAction≫ Modify`).

This annotation is taken into account in the **IF** statement. Then rule R3 is a predicate which conditions the granting of the execution of action *MedicalRecord__SetData* (a setter of the *data* attribute) and which means that the doctor must be employed by the current hospital of the patient:

```
P(user,instance) ≙
    (user ∈ Doctor ⇒ HospitalDoctorRel(user) =
    PatientHospitalRel(PatientMedicalRecordRel(instance)))
```

This constraint shows the impact of the functional model on the access control model because the hospital in which the patient is admitted and the hospital of the connected doctor originate from the functional model. In the tool translating SecureUML models into B machines, this kind of annotation is taken into account and inserted with no modification in the B operation. This requires that the designer expresses the constraint as B statements. In SecureUML these constraints can be modeled using OCL, however there is no tool that can validate both static and functional models with such constraints [22].

Fig. 2.9 details the operation that check if the action *GetData* on a medical record is granted: *secure_MedicalRecord__GetData*. This operation is another example of the translation of the SecureUML model into B.

```
answer ← secure_MedicalRecord__GetData(Instance, user, role) ≙
   PRE
      Instance ∈ MedicalRecord ∧
      user ∈ USERS ∧ role ∈ ROLES
   THEN
      IF
         role ∈ roleOf(user)
      THEN
         answer := granted
      ELSE
         answer := denied
      END
   END
```

Figure 2.9: Operation *secure_MedicalRecord__GetData*

2.4.4 Dynamic Access Control Model Translation

In [29] we have specified translation rules from ASTD to Event-B. However, one goal of the Selkis project is to implement an access control filter for information systems. Thus, we need the refinement process of the B method that leads to a proved implementation. In order to do so, we have adapted translation rules of [29] for B as described in [28]. This translation is made in three steps. The first step starts from the root ASTD and goes down to the leaves: a variable is created for each ASTD in order to encode its state. The second step creates a B operation for each transition label. In this step, a set of constant B functions is also created to encode the transitions of all the automata. Each transition label of each automaton has its own B transition function. Finally, the third step starts from the leaves ASTD and goes up to the root ASTD. It modifies the B operations according to the semantics of the type of each nested ASTD.

Fig. 2.10 presents the B operation for the action *Dynamic_MedicalRecord__GetData*. This action is affected by rule R4 described by the ASTD presented in Fig.2.3. The B translation of the ASTD generates several conditions that refer to the current state of the IS. The first two conditions $user \in Doctor$ and $role = DoctorRole$ ensure that the user willing to read the medical record is currently a doctor and is connected with the role *DoctorRole*. The third condition is the translation of the guard on the action of the ASTD as presented in Fig.2.3 ; it ensures that $\mathsf{joinHospital}(d, h)$ was executed, hence the value of the hospital h for the doctor d has been recorded by the system thanks to the $StateQchoice$ function associated with the quantified choice operator of ASTD. The next condition

$$StateAutomaton\_DoctorHospital(user) = dp$$

ensures that the doctor did not leave the hospital by checking that the ASTD is still in the state dp, *i.e.* before the execution of $\mathsf{leaveHospital}(d, h)$. Finally, the last condition

$$StateAutomaton\_dp(user, Instance) = p1$$

ensures that the patient was indeed admitted in the hospital after the doctor has joined the hospital. $StateAutomaton\_dp$ and $StateAutomaton\_DoctorHospital$ are partial functions used to model respectively the state of the inner automaton and the state of the upper automaton, in which dp is a place. Complete description of these functions is provided in [28].

2.5 Specification of the Access Control Filter

Once the different kinds of access control rules have been specified, it is necessary to define how they are combined and how the final decision of permitting the execution of an action by a specific user is

```
answer ← Dynamic_MedicalRecord__GetData(Instance,user,role ) ≙
  PRE
    Instance ∈ MedicalRecord ∧
    user ∈ USERS ∧
    role ∈ ROLES
  THEN
    LET pp BE pp = PatientMedicalRecordRel (Instance) IN
      IF
        user ∈ Doctor ∧
        role = DoctorRole ∧
        StateQchoice(user) = PatientHospitalRel(pp) ∧
        StateAutomaton_DoctorHospital(user) = dp ∧
        StateAutomaton_dp(user,pp) = p1
      THEN
        StateAutomaton_dp(user,pp) := p1 ||
        answer := granted
      ELSE
        answer := denied
      END
    END
  END
```

Figure 2.10: Operation *Dynamic_MedicalRecord__GetData*

calculated. This is the role of the access control filter. We use the B refinement process to specify it. First an abstract B model is built and its refinement allows the decision algorithm to be described.

2.5.1 Abstract Access Control Filter Specification

The abstract filter B machine is built by creating one B operation for each action to secure in the IS plus one for the rollback of the system. The rollback operation (presented in Fig. 2.11) is needed in the case where access control grants the execution of one action that cannot be executed by the functional part of the system. Hence, dynamic access control filter that depends on the state of the IS must be restored to a state where the action has not been executed. The other operations describe inputs and outputs for the filter's operations. An example of these operations is presented in Fig. 2.11 with the filtered version of the B operation *Filter_MedicalRecord__GetData*. At this level of specification, we only describe the goal of the operation *i.e.* granting or denying the execution of the action. Substitution *CHOICE* is a non deterministic choice between three substitutions. The algorithm that calculates which answer will be returned is detailed in the next step, the refined filter specification.

2.5.2 Refinement of the Access Control Filter

The abstract access control filter is refined into a more concrete filter that includes static, dynamic and functional models. The inclusion of these machines permits the execution of operations from them. The refinement of these operations introduces calls to static and dynamic access control operations ; if the policy grants the execution, the functional operation is executed. Fig. 2.12 presents the overall view of the approach, with the graphical specifications translated into B machines combined in an access control filter. The top layer presents the different models expressed using graphical notations that are then translated into B machines. The second part of Fig. 2.12 introduces the abstract access control policy specification composed by all the translated B machines.

```
$Rollback() \triangleq$
    PRE
        $rollbacklock = locked$
    THEN
        $rollbacklock := unlocked$
    END

$answer, executed, data$
  $\leftarrow Filter\_MedicalRecord\_\_GetData(Instance, user, role) \triangleq$
    PRE
        $Instance \in MedicalRecord \wedge$
        $user \in USERS \wedge$
        $role \in ROLES \wedge$
        $rollbacklock = unlocked$
    THEN
        CHOICE
            $answer := granted \parallel executed := ok$
            $\parallel data :\in THEDATA \parallel rollbacklock := unlocked$
             OR
            $answer := granted \parallel executed := notExecuted$
            $\parallel data :\in THEDATA \parallel rollbacklock := locked$
             OR
            $answer := denied \parallel executed :\in \{ok, notExecuted\}$
            $\parallel data :\in THEDATA$
            $\parallel rollbacklock :\in \{locked, unlocked\}$
        END
    END
```

Figure 2.11: Abstract operations *Rollback* and *Filter_MedicalRecord__GetData*

The access control policy designer has to specify the algorithm that computes the answer of his/her policy. We call this feature the decision algorithm. An acceptable decision algorithm is to put static and dynamic filters in conjunction. This means that both static and dynamic access control policies must grant the execution in order for the filter to grant the execution. In our example, we have chosen this solution, but any other combination can be specified. We could imagine another way to combine policies for our example: in the case of emergency, the dynamic access control policy could be replaced by a new one, more permissive, in order to reduce constraints and improve efficiency of medical staff. Fig. 2.13 is the refinement of operation *Filter_MedicalRecord__GetData* introduced in Fig. 2.11 and describes the combining algorithm for static and dynamic policies.

Rollback is an important part of our filter. Contrary to the static access control specification that does not evolve when executed, the dynamic access control specification is based on a state that must be consistent with the state of the IS. In the case where both static and dynamic policies grant the execution and that the execution fails at the functional level, we have to restore the state of the dynamic policy to its previous state. We do not detail the refinement of the rollback operation, but it consists in calling a B operation of the dynamic access control B specification in order to restore it to its previous state. However this requires that the previous state was saved into variables before. In order to do so, we have defined an operation called *saveDynamicState* in the dynamic access control that will backup state variables before any call to operations.

Such an access control filter can be implemented using the BPEL language [2] and can be included in a service oriented architecture (SOA) environment [7]. When IS are implemented using Web services,

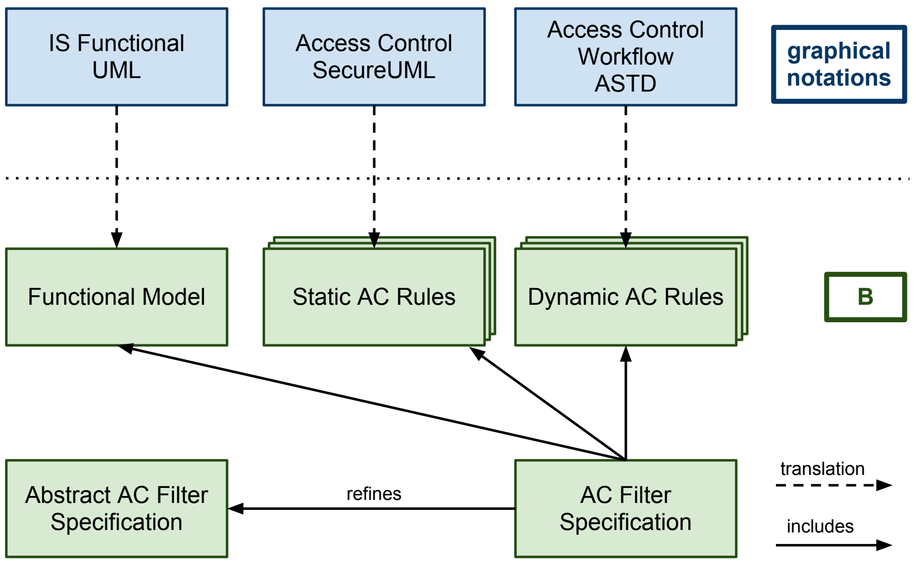


Figure 2.12: Overall view of the access control filter

like in SOA, security features are often implemented in a Policy Enforcement Manager (PEM). A PEM is based on two main parts: the Policy Decision Point (PDP) and the Policy Enforcement Point (PEP). The PDP takes the decision to grant or deny the execution based on several informations such as the policy and combining algorithms. The PEP is the link between the PDP and the functional IS. Our filter perfectly corresponds to a PEM since it provides exactly the functionalities of both a PEP and a PDP.

2.5.3 Verification and validation purpose

The various graphical diagrams are dedicated to a better understanding of several aspects of an information system and their corresponding formal models support a rigorous reasoning about these aspects. For our case study, we proved the machines and the refinement using Atelier-B prover. Validation can be performed by animating the model. This consists of playing several scenarios with the ProB [23] tool:

Normal scenario

- A secretary S of a hospital H creates a patient P, admits him in the hospital and associates to him an empty medical record;
- A doctor D joins the hospital H;
- The doctor D modifies attribute data of the medical record of P and validates it;
- The patient P leaves the hospital H.

Attack scenarios

- A secretary trying to validate a medical record;
- A doctor trying to modify a medical record of a patient who belongs to another hospital;
- A patient who leaves a hospital with a medical record which is not validated.

It is also possible to verify properties against the whole system, *i.e.* the combination of the filter, the access control machines and the functional machine. We can verify that temporal properties hold using model checkers such as ProB [23] or using the proof-based verification approach developed by Mammar et al. in [26] and Frappier et al. in [12]. For instance, we could verify that rule R5 is correctly enforced by checking that the CTL property

$$\mathbf{AG}\ (\neg\ MedicalRecord\_isValid(i) \Rightarrow$$

$$\mathbf{AGEF}\ pre(MedicalRecord\_Validate(i)))$$

```
answer, executed, data
  ← Filter_MedicalRecord__GetData(Instance, user, role) ≙      PRE
      Instance ∈ MedicalRecord ∧
      user ∈ USERS ∧
      role ∈ ROLES ∧
      rollbacklock = unlocked
    THEN
      VAR static, dynamic, functional, return IN
        static ←
         secure_MedicalRecord__GetData(Instance,user,role) ;
        IF static = granted
        THEN
          dynamic ←
            dynamic_MedicalRecord__GetData(Instance,user,role) ;
          IF dynamic = denied
          THEN
            answer := denied ; executed := notExecuted ;
            data :∈ THEDATA
          ELSE
            functional, return
              ← MedicalRecord__GetData(Instance) ;
            IF functional = ok
            THEN
              answer := granted ; executed := ok ; data := return
            ELSE
              rollbacklock := locked ;
              answer := granted ; executed := notExecuted ;
              data :∈ THEDATA
        ELSE
          answer := denied ; executed := notExecuted ;
          data :∈ THEDATA
      END
    END
```

Figure 2.13: Refinement of *Filter_MedicalRecord__GetData*

holds against the specification of the filter. Indeed, the predicate $\neg$ *MedicalRecord_isValid*(i) means that i, an instance of the *MedicalRecord* class, has been modified but not yet validated. Informally, this property means that if an instance of the *MedicalRecord* class has been modified but not yet validated (predicate $\neg$ *MedicalRecord_isValid*(i)) it is always possible to validate it. In other words, since we want to ensure that after one or more modifications, the medical record will be validated, then we want to check that on all execution paths we can always find a path that eventually leads to a state where the precondition of the B operation *MedicalRecord_Validate*(i) holds.

2.6 Conclusion

In our work, we apply a combination of formal and graphical techniques in order to propose a technique covering all aspects of access control policies in the context of Information Systems. Our methodology starts by various kinds of graphical models and produces a complete formal B specification. We use UML class diagrams to model structural functional models, SecureUML to express static access control rules and ASTD to represent history-based rules. In order to remedy to the lack of tools for verifying or analyzing (formally) these diagrams we translate them into B. Our work is then intending to formalize access control policies in order to reason on the derived formal specifications using associated tools: AtelierB prover and ProB model checker and animator [23, 24]. Another contribution of our approach is that it becomes possible to design information systems as a whole using graphical views for its functional and access control aspects, and then generate a complete formal specification for the whole system.

Works on OrBAC [6] propose procedures to analyze security policies, however they do not take into account functional models. Links between access control rules and the functional specification cannot be formally checked. This is done rather in the implementation or deployment steps. In [25], authors have conducted an in-depth analysis of the literature on research works that combine graphical and formal methods for designing IS, including both functional and access control purposes. To our knowledge, the closest work to ours is presented in [37], where functional and security models are merged into a single UML model which is translated into Alloy. However, the access control rules described in [37] are mainly of static nature. Moreover Alloy proposes verification techniques based on model-checking whereas B also provides a theorem prover.

We are currently working on the tool implementing the translation of ASTD into B. Further work includes the evaluation of our approach on case studies of the Selkis Project.

Chapitre 3

Un méta-modèle B pour l’expression de politiques de contrôle d’accès

Dans ce chapitre, nous nous intéressons à la définition d’un méta-modèle étendant le méta-modèle B permettant de définir un filtre de contrôle d’accès. Cet article doit être considéré comme venant en complément de celui présenté dans le chapitre 2. Il décrit un méta-modèle permettant de définir les concepts importants d’une politique de contrôle d’accès en B en prévision du raffinement qui conduira vers l’implémentation.

La vérification et la validation d’une politique de contrôle d’accès pour un système d’information est une tache difficile mais nécessaire. Afin de tirer avantage des caractéristiques formelles et des outils de la méthode B, nous introduisons dans cet article un méta-modèle de modélisation B pour les politiques de contrôle d’accès. Ce méta-modèle est une base nécessaire pour le développement d’un prototype formel d’un filtre de contrôle d’accès combiné au système d’information, afin de vérifier et valider une politique avant son implémentation.

Une première version de cet article a été soumise et acceptée à la quatrième édition du workshop international *Foundations & Practice of Security* (FPS 2011) ayant eu lieu à Paris en France du 12 au 13 mai 2011. Nous présentons ici une version étendue qui fait l’objet d’une publication dans la série *Lecture Notes of Computer Sciences* de Springer dans le volume 6888.

A Metamodel of the B Modeling of Access-Control Policies

Jérémy Milhau [1,2], Marc Frappier [1], Régine Laleau [2]

1 : GRIL, Université de Sherbrooke, 2500 Boulevard de l'Université, Sherbrooke QC, Canada
{Jeremy.Milhau,Marc.Frappier}@usherbrooke.ca

2 : LACL, Université Paris-Est, IUT Sénart Fontainebleau, Département Informatique
Route Forestière Hurtault, 77300 Fontainebleau, France
{Laleau}@u-pec.fr

Abstract Verification and validation of access-control policies for information systems is a difficult yet necessary task. In order to take advantage of the formal properties and tools of the B method, we introduce in this paper a metamodel of the B modeling of access control policies. This metamodel leads to the development of a formal prototype of an access control filter combined to the system. It allows verification and validation of policies before implementation.

Keywords Metamodel, access-control, IS, formal method, MDA, B [1]

3.1 Introduction

In the context of information systems (IS), we advocate the use of formal methods in order to prevent unexpected behavior and produce a software correct by design. Following this approach, we have proposed a way to specify the functional part of an IS and to systematically implement it using the APIS platform [11]. We now focus on the specification of access-control (AC) policies, *i.e.* the expression of rules describing if an action can be executed in the system given a context. We think that separating AC policies from the functional core of a system fosters maintainability and reduces the complexity of future modifications.

However, expressing AC rules is not a trivial task. Several languages and notations have been proposed, each one with its strengths and weaknesses. Our approach encompasses two steps to specify AC rules. The first one consists in expressing AC rules with UML-like graphical notations, thus they can be understood and validated by stakeholders. Then these graphical notations are translated into formal notations in order to be verified and animated [30].

1. This research is funded by ANR (France) as part of the SELKIS project (ANR-08-SEGI-018) and by NSERC (Canada).

3.1.1 Combining Static and Dynamic rules

AC rules can be characterized by several criteria. For instance, they can be either static or dynamic. According to Neumann and Strembeck in [31], dynamic rules are defined as rules that can only be evaluated at execution time according to the context of execution. Static rules are defined as invariants on the state of a system. For example a static rule would grant the execution of action *a* by user *u* at any time, whereas a dynamic rule would grant the execution of action *a* by user *u* only if action *b* was not executed by *u* before. In order to be evaluated, such dynamic rules require an history of previously executed actions, which can be represented in either the IS state or the policy enforcement manager state. They define allowed or forbidden workflows of actions in a system. We have chosen two notations: Secure-UML and ASTD in order to model AC policies [30].

3.1.2 A Formal Notation for AC rules

The B notation, developed by Abrial [1], is an abstract machine notation based on set theory and first order logic. It involves a development approach based on refinements of models called *machines*. The initial machine is the most abstract model of the system. Each subsequent refinement step introduces more concrete elements into the model until a final refinement produces an *implementation*, *i.e.* a machine that can be implemented using a programing language such as C. Each machine is composed of static and dynamic parts. The static part refers to *constants* and *invariants*, *i.e.* properties of the machine. The dynamic part refers to *variables* and *operations* that modify variables of the machine.

We would like to take advantage of verification and validation tools associated with the B method such as ProB [23], an animator and model checker for B specifications. For that reason, we have chosen to use B to express AC policies.

In order to express dynamic rules in a platform-independent model (PIM), we used the ASTD notation [13], an extension of Harel's Statecharts using process algebra operators. This notation offers an automata-like representation of workflows which can be combined using sequence, guard, choice, Kleene closure, interleave, synchronization and quantifications over choice and interleave. It also provides a graphical representation of dynamic rules. For static rules, we use a SecureUML-based notation [25]. We then translate both ASTD specifications [29] and SecureUML diagrams into B machines (named *Dynamic_Filter* and *Static_Filter* respectively in the following sections). The final step of our work is to build an AC filter implemented using the **BPEL!** (**BPEL!**) notation [9]. We want to use the refinement approach of B to obtain the implemented filter. In order to do so, we have to identify the concepts required in the implementation and to extend the B metamodel with these new *types* for AC modeling. We then provide refinement rules between the new types and their implementation.

3.2 A Proposal for AC Modeling using B

Our proposed metamodel is an extension of the B metamodel [17] with AC aspects. It describes an AC engine that is able to grant or deny the execution of an action of the IS according to an AC policy. If certain conditions are met then the action is executed. If one of the conditions does not hold, then it denies the execution. Conditions, *i.e.* AC rules, can refer to security parameters (such as user, role, organization, . . .) or functional (business) parameters.

Our goal is the refinement to several platform-specific models (PSMs). In order to do so, we have to specify concepts such as rules or policies that are not explicitly specified in the original B metamodel. In our PIM metamodel, we specialize classes from the B Metamodel into new ones that correspond to concepts used for defining AC policies. This helps the refinement process by linking a concept of the abstract level to a concept of a more concrete level. The metamodel also combines the static and dynamic parts of the specification of an AC policy. Since the metamodel is composed of many classes and is quite large, we will describe it here in a fragmented way.

3.2.1 Combinaison of B machines

Fig. 3.1 presents the part of the metamodel that describes machines and operations used to model a complete AC engine. We will detail in the following sections this part of the metamodel from top to bottom. The first line is composed of B machines as denoted by the generalization link between classes. The AC engine is composed of at least four machines. Each one plays a role in the process of granting or denying the execution. The four kinds of machines composing an engine are the following:

1. The functional machine is the core of the IS. It describes how the system evolves over time and how attributes and classes of the system are modified by the execution of an action. The functional model of the system can also introduce ordering constraints on the actions to be executed.
2. Several *static filters* can be included in the engine. Each one of them describes a static policy, *i.e.* a set of static AC rules. Such rules are expressed in an RBAC-like (Role based access-control) notation. For example, there may be one static filter for authorization and one for prohibition.
3. Several *dynamic filters* can be included in the engine. Each one of them describes a dynamic policy, *i.e.* a set of dynamic AC rules that can grant or deny an execution according to the state of the IS, or the state of the dynamic policy itself. All dynamic filters may not be needed at all time. One of the dynamic filters can be activated, for example, in emergency situations only.
4. The AC filter is the main part (controller) of the engine. It is in charge of asking each machine if the action can be executed according to several parameters. It includes all the other machines so it can call their operations. Then it combines the answers it received in order to decide if the action is granted or not. According to the context, it can decide to ignore the decisions made by the static filters and the dynamic filters, for instance in the case of emergencies in an hospital.

Our metamodel also takes into account operations of these B machines. To illustrate them, we use a case study of an hospital. We consider an action of the functional part of the system, called $\mathsf{Admission}(\,Patient\,)$, that denotes the admission of a patient in one of the units of the hospital. Such an action is generally performed by a user of the IS who is a doctor.

1. The functional operation $\mathsf{Admission}(\,Patient\,)$ is in the functional machine. It is an example of a *Functional_Action*. It specifies all the modifications that the execution of the action performs on all the entities of the IS and their attributes. It can also describe conditions in order for the action to be executed, for example the patient must have paid his medical bills before being admitted.
2. The static operation $\mathsf{Static\_Admission}(\,Patient, User\,)$ is in the static machine. It is an example of a *Static_Action*. It will return *granted* or *denied* according to the static policy. Note that in our example a security parameter *User* was added, compared to $\mathsf{Admission}(\,Patient\,)$. This addition can be used to check that the user is a doctor and not a nurse for instance.
3. The dynamic operation $\mathsf{Dynamic\_Admission}(\,Patient, User\,)$ is in the dynamic machine. It is an example of a *Dynamic_Action*. It will return *granted* or *denied* according to the dynamic policy. The security parameter *User* can be useful if we want to check that the user of the IS (that should be a doctor, according to the static policy) is a doctor in the same hospital as the patient.
4. Finally, the operation $\mathsf{AC\_Admission}(\,Patient, User\,)$ is in the AC filter machine. It is an example of an *AC_Action*. It calls all static and dynamic operations described above and computes wether or not to execute the action $\mathsf{Admission}(\,Patient\,)$ of the functional machine. In our example, the algorithm can take into account emergency situations and the filter will bypass the static policy if, for instance, an emergency is declared.

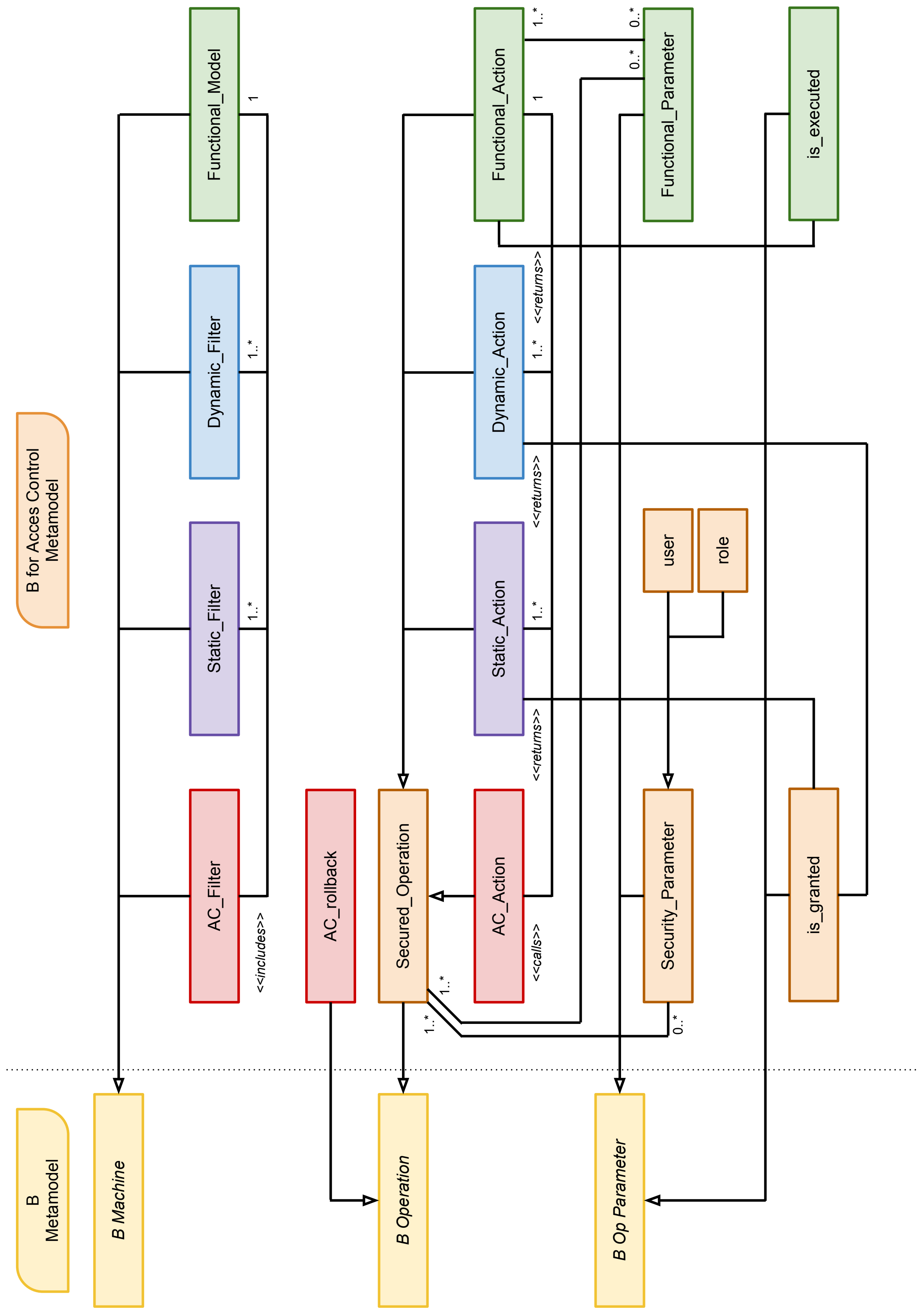


Figure 3.1: Main part of the metamodel for AC modeling in B

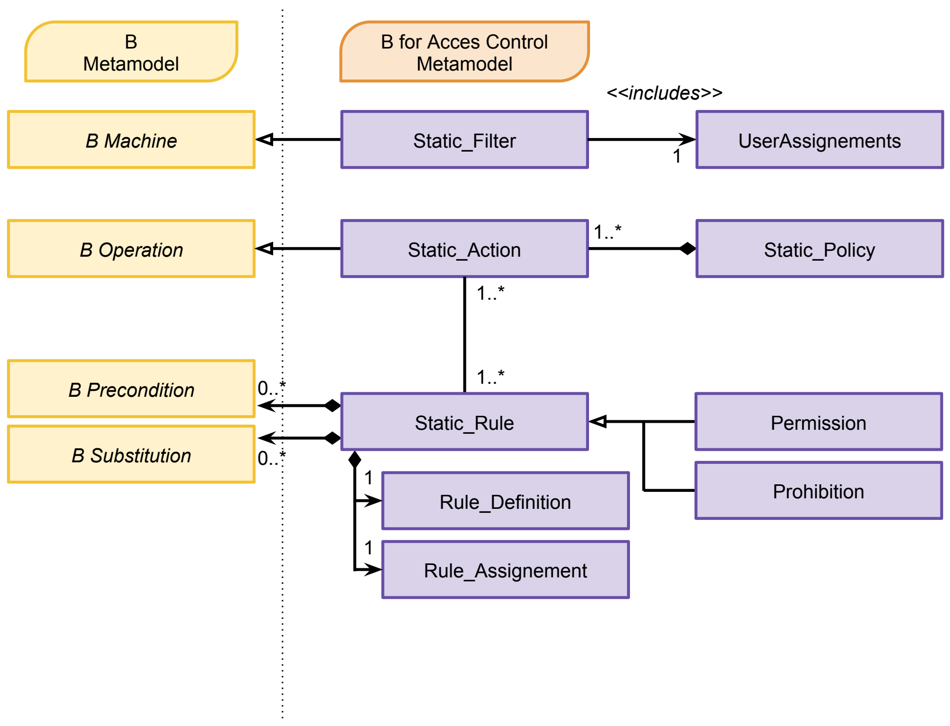


Figure 3.2: Details of the static machine of the metamodel for AC modeling in B

We also need another operation called rollback( ) that is part of the AC filter machine. This operation may be required in specific cases. For instance, if an action was granted to be executed by all static and dynamic machines but the execution fails in the functional machine, the rollback operation must be called. Indeed, the dynamic policy state is modified when the answer *granted* is given. In the event of the failure of the execution in the functional machine, the previous state of the dynamic policy must be restored. This is the role of operation rollback.

Operation parameters are either security parameters or functional parameters. Security parameters refers to user, their role and any other non-functional related information, whereas functional parameters are parameters of the actions of the IS. All operations from static and dynamic machines return *granted* or *denied* as represented by the *is_granted* class and functional operations return *success* or *failure*.

3.2.2 The static filter

The static filter machine defines a static AC policy that will be enforced on the system. In our example, we have chosen an RBAC-like policy. Each rule of such policy can be expressed as a permission or a prohibition. A rule is a combination of instances of role and action classes. For instance, in an hospital, we can give the permission to any user connected as a doctor to perform action Admission(*Patient*).

In Fig. 3.2 the class *Rule_Definition* defines either a permission to perform an action or a prohibition. Such a definition is then linked to a role by the class *Rule_Assignement*. The combinination of both is called *Static_Rule* and can be used in one or more B operations (*Static_Action*). The set of all *Static_Action* define the *Static_Policy* of the *Static_Filter* machine. The *Static_Filter* includes the machine *UserAssignements* that defines a role hierarchy and the the roles that a user can play.

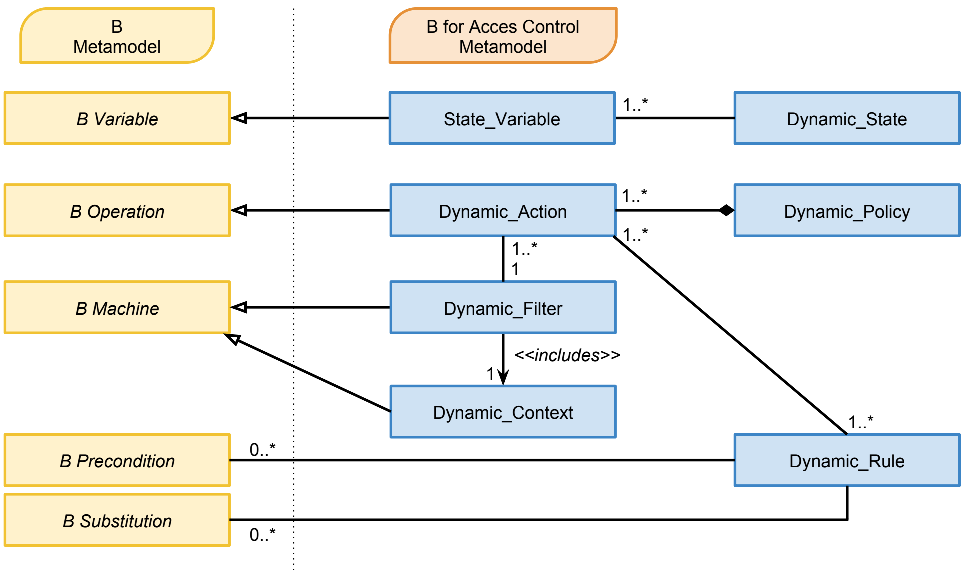


Figure 3.3: Details of the dynamic machine of the metamodel for AC modeling in B

3.2.3 The dynamic filter

The dynamic filter machine defines a dynamic AC policy that will be enforced on the system. Such policies can be defined using the ASTD notation. When describing an ASTD, there are several parts to take into account. An ASTD is composed of a topology and a state. The topology refers to the structure of the ASTD. It is unaffected by the execution of actions. On the contrary, the state of the ASTD evolves each time an action is executed.

In order to model a workflow using the B formalism, we must encode the topology and the state. We do so in our metamodel as depicted by Fig. 3.3. The state of the workflow (*Dynamic_State*) is stored into several variables called *State_Variable*. The *Dynamic_Policy* is composed of several *Dynamic_Actions* encoding the topology of the workflow into *Dynamic_Rules*. Each rule is composed of B preconditions that can test the value of several *State_Variables* and B substitutions that will modify them, hence the evolving *Dynamic_State*.

3.3 Conclusion

We have defined a metamodel allowing to use the B notation to model access-control policies to be enforced upon an information system. Such B model can be used in order to validate the policy and perform verification and proof of properties, improving the trustworthiness and security of such information systems. In order to validate our approach we have implemented our metamodel using a medical IS case study [30].

The ORKA project [4] proposed a framework to compare the features of AC methodologies. Each method is based on at least one notation that is very good at expressing one type of rule (*i.e.* static rules for *RBAC, dynamic rules for ASTD, etc.). But expressing other types of rules may be difficult (as for static rules with ASTD) or requires the coding of a context introducing state variables. Our approach solves this problem by providing adapted languages for each rules. The combining of all parts of the policy into one single formal model also helps to validate the entire specification. The ORKA project evaluates methods according to criteria such as formalism of the notation, availability of verification and validation tools, availability of a graphical notation for the policy and the possibility to express constraints. Our approach was developed in order to have all these features available.

We are currently working on the next step, *i.e.* translating a model using the refinement mechanisms of B. This will ensure that a correct implementation of AC policies is provided and improve reliability on such a critical part of the system. We are also working on tools for the translation steps of our approach.

Chapitre 4

Une stratégie de raffinement pour l'implémentation de politiques de contrôle d'accès

Dans le chapitre 2 nous avons introduit un modèle formel de filtre de contrôle d'accès que nous avons ensuite détaillé dans le chapitre 3 en définissant un méta-modèle MMB_{sec}, extension du méta-modèle B pour identifier les concepts B correspondant à un filtre de contrôle d'accès. Dans ce chapitre, nous nous intéressons à la définition d'une stratégie de raffinement B permettant d'obtenir l'implémentation B d'un filtre de contrôle d'accès.

4.1 Introduction

Dans le cadre du projet Selkis, plusieurs implémentations d'un filtre de contrôle d'accès sur des architectures différentes ont été développées. Afin de préciser les concepts nécessaires à la définition d'une implémentation d'un filtre de contrôle d'accès, un méta-modèle appelé MM_{imp} a été défini [8]. Il décrit l'architecture d'un filtre de contrôle d'accès dans le cadre d'une architecture orientée services (SOA). Ce méta-modèle a été développé par Michel Embe Jiague *et al.* en collaboration avec les partenaires industriels du projet Selkis. L'objectif de ce chapitre est de définir une stratégie de raffinement B du filtre de contrôle d'accès afin d'obtenir une implémentation qui raffine la spécification abstraite de la politique de contrôle d'accès et qui respecte MM_{imp}. Cette implémentation B peut ensuite être traduite dans une implémentation du filtre de sécurité respectant l'architecture désirée. Nous avons choisi de traduire cette implémentation en BPEL pour pouvoir la comparer à celle de Embe Jiague [7] qui est également réalisée en utilisant BPEL dans une architecture SOA. Cette dernière implémentation est obtenue par la traduction directe de la spécification ASTD en BPEL, sans passer par B.

Dans la section 4.2, nous étudions le méta-modèle d'implémentation MM_{imp} proposé par Embe Jiague *et al.* et ses liens avec le méta-modèle MMB_{sec} décrit dans le chapitre précédent. Nous définissons une stratégie de raffinement qui permet de se conformer à ce méta-modèle dans la section 4.3. Enfin dans la section 4.4, nous proposons les grandes lignes d'une traduction de l'implémentation B en BPEL.

Dans la suite de ce chapitre, nous nous intéressons à l'implémentation B de la machine *AC_filter* ainsi que l'implémentation des autres machines qu'elle inclut, telles que définies dans le chapitre 2 à la Figure 2.12. Ces implémentations B s'obtiennent par raffinement des machines, en levant le caractère non déterministe des substitutions utilisées ainsi qu'en procédant à un raffinement des structures de données. Les machines à raffiner sont :

1. Le filtre de contrôle d'accès : la machine *AC filter Specification* dans la Figure 2.12, *AC_filter* dans MMB_{sec}
2. La machine décrivant les aspects fonctionnels : *Functional_Model* dans la Figure 2.12 et dans MMB_{sec}
3. La machine décrivant la politique statique, issue de la traduction de la spécification SecureUML : *Static AC Rules* dans la Figure 2.12, *Static_filter* dans MMB_{sec}
4. La machine décrivant la politique dynamique, issue de la traduction de la spécification ASTD : *Dynamic AC Rules* dans la Figure 2.12, *Dynamic_filter* dans MMB_{sec}

Pour l'implémentation de la machine décrivant les aspects fonctionnels, nous pouvons utiliser le processus couvert par les travaux de Mammar *et al.* [27]. Les auteurs proposent la génération de transactions SQL et le code Java associé pour les spécifications B des applications bases de données. Il est possible d'utiliser, en les adaptant, les travaux présentés dans [3] où l'auteur détaille l'implémentation B de politiques de contrôle d'accès de type RBAC et définit des obligations de preuve permettant de garantir que l'implémentation respecte bien la spécification de la politique de contrôle d'accès. Dans la suite de cette section, nous nous intéressons donc à l'implémentation du filtre de contrôle d'accès et à l'implémentation de la machine B décrivant le comportement dynamique de la politique issue de la traduction des ASTD en B.

4.2 MM_{imp} : Un méta-modèle d'implémentation de politiques de contrôle d'accès

Dans [8], un méta-modèle MM_{imp} pour un PEM (*Policy Enforcement Manager*) gérant le contrôle d'accès dans une architecture SOA a été présenté. Ce méta-modèle définit les composants requis pour le PEM. Les auteurs donnent également les diagrammes de séquence décrivant le comportement du PEM. La Figure 4.1 décrit l'ordonnancement des actions sur chaque élément du filtre de contrôle d'accès

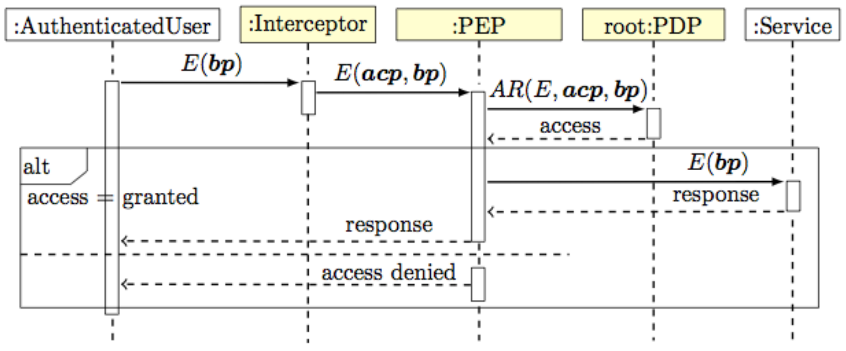


FIGURE 4.1 – Diagramme de séquence : exécution de l'action $E(bp)$ par l'utilisateur

avant l'appel du service, *i.e.* l'aspect fonctionnel du système. Dans ce diagramme de séquence, plusieurs acteurs sont mentionnés. Un utilisateur authentifié (*:AutenticatedUser*) envoie une requête notée $E(bp)$ comprenant des paramètres métiers, notés bp. Un intercepteur (*:Interceptor*) récupère cette requête avant sa transmission au service concerné, la complète en ajoutant les paramètres de sécurité (acp) et envoie une nouvelle requête ($E(acp, bp)$) au PEP (*Policy Enforcement Point* noté *:PEP* dans le diagramme). Le PEP effectue alors une requete d'autorisation notée $AR(E, acp, bp)$ pour l'action E et ses paramètres bp ainsi que les paramètres de sécurité acp auprès du PDP principal (*:rootPDP*). Une fois la réponse du PDP connue par le PEP, il transmet la requête initiale de l'utilisateur au service si le PDP l'autorise. Il indique à l'utilisateur que sa requête est refusée si le PDP n'a pas autorisé l'exécution.

Dans le cadre des machines B modélisant un filtre de contrôle d'accès, l'*Interceptor*, le *PEP* et *root :PDP* correspondent à la machine B *AC_filter*. En effet, cette machine reçoit la requête de l'utilisateur $E(bp)$, lui ajoute les paramètres de sécurité et va consulter le filtre qui retournera alors la réponse de la politique. Si l'accès est autorisé, la machine transmet la requête $E(bp)$ de l'utilisateur à la machine fonctionnelle (*Service* dans le diagramme de séquence) qui lui retourne sa réponse. Si l'accès est refusé, l'utilisateur est notifié.

Le diagramme de séquence présenté dans la Figure 4.2 détaille le processus de décision conduit par le PDP. Lorsque le PDP principal reçoit une requête d'autorisation $AR(E, acp, bp)$, il la transmet à un des PDP esclaves (*slave :PDP*). Ces PDP sont facultatifs, ils peuvent être utilisés lorsque la politique peut changer en fonction du contexte. Les PDP esclaves sont utiles par exemple quand dans un hôpital en situation d'urgence les contraintes de la politique de contrôle d'accès sont temporairement levées. Un PDP esclave permissif est alors mis en place, et se substitue au PDP esclave habituel. Cet exemple n'est évidement pas restrictif, et nous pouvons imaginer d'autres utilisations pour ces multiples PDP esclaves. Le PDP esclave qui a reçu la requête d'autorisation consulte alors l'ensemble des filtres. Leurs réponses sont combinées et envoyées au moteur de décision (*:DecisionEngine*) qui répond à la requête d'autorisation positivement si la politique autorise l'exécution de $E(bp)$ dans le contexte de sécurité acp.

Dans le cadre des machines B modélisant un filtre de contrôle d'accès, il n'y a qu'un *slave :PDP* chargé de consulter les différents filtres (les machines *Dynamic_filter* et *Static_filter*) et combiner leur réponse avant de déclarer si la politique autorise ou non l'exécution. Ce rôle est confié à la machine B *AC_filter*. Cette machine est également le moteur de décision puisque dans chacune des opérations B, une substitution **IF** décrit l'algorithme de calcul de la décision de la politique de contrôle d'accès pour une des actions du système en fonction des valeurs de retour des opérations des machines *Dynamic_filter* et *Static_filter*.

La Figure 4.3 fait le lien entre le diagramme de classes décrivant le filtre dynamique du méta-modèle MM_{imp} et les classes issues du méta-modèle MMB_{sec} présenté à la Figure 3.3. Les pointillés rapprochent les classes exprimant des concepts similaires. Ainsi, le filtre du méta-modèle MM_{imp} (noté *filter* dans le diagramme *dynAC*) correspond à notre machine *Dynamic_Filter*. La politique dynamique (*Policy*)

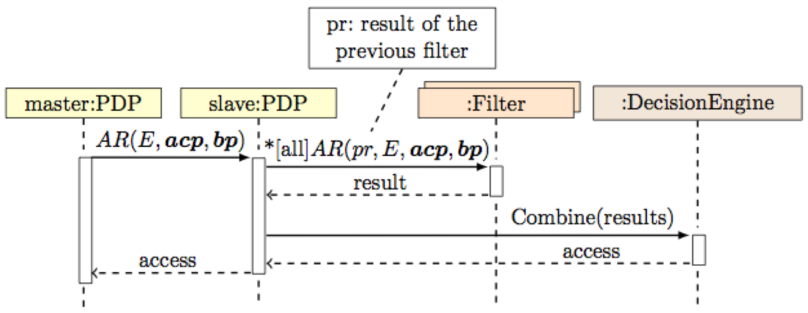


FIGURE 4.2 – Diagramme de séquence : prise de décision en consultant les filtres

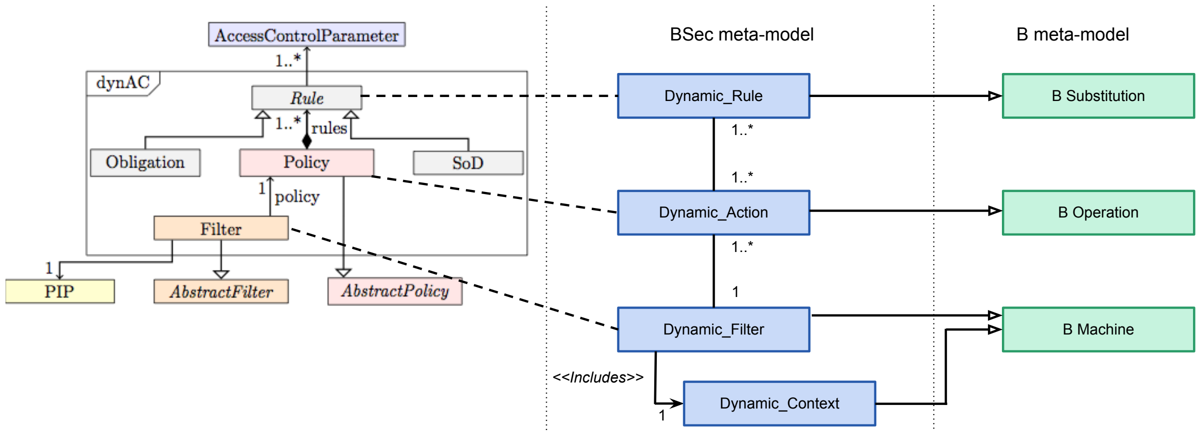


FIGURE 4.3 – Rapprochement du méta-modèle MM_{imp} et du méta-modèle MMB_{sec} : partie dynamique

correspond à l'ensemble des opérations B (*Dynamic_Action*) de la machine *Dynamic_Filter*. Enfin, une règle (*Rule*), correspond à une substitution B (*Dynamic_Rule*) d'une opération B (*Dynamic_Action*).

La Figure 4.4 résume les liens entre les classes *AC_Filter* et *Dynamic_Filter* du méta-modèle MMB_{sec} présenté à la Figure 3.1 et les acteurs des diagrammes de séquence de Embe Jiague *et al.*. Un rapprochement entre les concepts décrits est modélisé par l'utilisation de pointillés. Ainsi, la machine B *AC_Filter* du MMB_{sec} correspond à la fois au *MasterPDP*, au *SlavePDP* ainsi qu'au *DecisionEngine* du diagramme de séquence. Le filtre dynamique correspond à un filtre dans le diagramme de séquences.

4.3 Raffinement du filtre de contrôle d'accès

Les contraintes sur les structures de données et sur les substitutions dans une implémentation B sont définies. Certaines substitutions sont interdites dans les implémentations comme par exemple les substitutions **PRE** ou **SELECT**. Concernant les structures de données typant les constantes, ensembles et variables, celles-ci doivent être implémentables, c'est-à-dire pouvoir être traduites en structures de données utilisables dans un langage de programmation impératif tel que C. Ainsi les ensembles abstraits doivent être représentés par de nouveaux ensembles comme par exemple des ensembles d'entiers. Les variables représentant un sous-ensemble d'un ensemble B sont modélisées par un tableau indicé par des entiers et contenant un booléen. Par exemple, soit E un ensemble abstrait contenant six éléments implémenté par l'ensemble 0..5, et soit v une variable abstraite typée comme étant un sous-ensemble

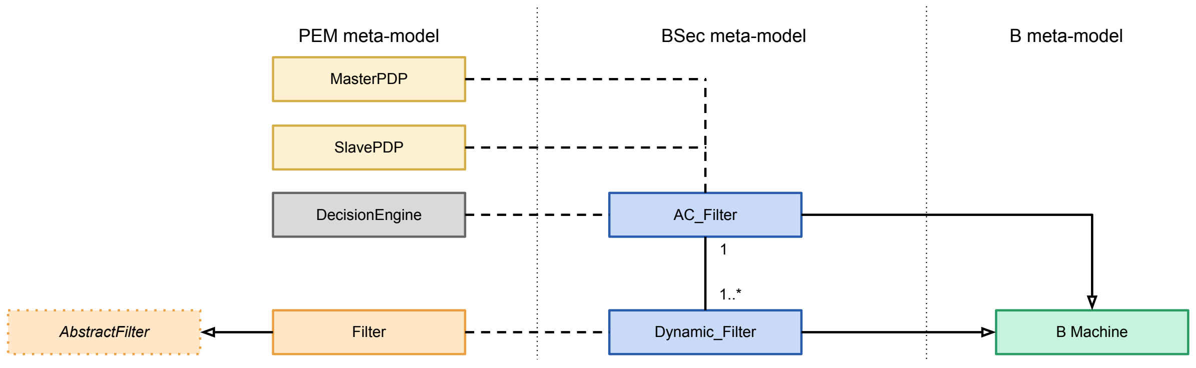


FIGURE 4.4 – Rapprochement des acteurs du diagramme de séquence de Embe Jiague *et al.* et du méta-modèle MMB_{sec}

de E. L'implémentation de v est une variable de type tableau, de taille 6, et contenant *true* dans les cases correspondant aux éléments abstraits de E qui sont élément de v. D'une manière similaire, seules les fonctions totales B dont l'ensemble de départ est implémentable peuvent être représentées par des tableaux. Il faut donc compléter les fonctions abstraites utilisées afin de les rendre implémentables.

4.3.1 Implémentation du filtre de contrôle d'accès : *AC_Filter*

La machine B modélisant un filtre de contrôle d'accès définie dans le chapitre 2 utilise des substitutions déterministes à une exception près. Dans le cadre d'un refus de l'exécution par la politique de contrôle d'accès, une valeur quelconque est retournée comme résultat de l'action *MedicalRecord__GetData*. Cette substitution est notée $data :\in THEDATA$ dans la Figure 2.13. Les autres substitutions utilisées à savoir **IF**, **VAR**, séquencement (**;**) et appels d'opérations conviennent pour une implémentation telle que décrite dans [1]. Lors du raffinement, il est alors nécessaire d'attribuer une valeur par défaut à la variable de retour *data* qui est choisie de manière déterministe. Par exemple, si l'ensemble $THEDATA$ est constitué de chaines de caractères de taille bornée, la chaine vide pourrait convenir comme valeur de retour en cas de refus de l'exécution de l'action *MedicalRecord__GetData* par le filtre de contrôle d'accès.

4.3.2 Implémentation du filtre dynamique : *Dynamic_Filter*

Nous allons maintenant étudier le raffinement des structures de données et des opérations générées par la traduction en B des ASTD.

Les ensembles de places Les ensembles de places sont codés en utilisant les entiers naturels pour représenter chaque place. Ainsi l'ensemble $NAME$ codant l'ensemble des noms des ASTD est représenté par l'ensemble $1..card(NAME)$. Chaque élément de $NAME$ devient donc un entier naturel de $1..card(NAME)$. L'ensemble $AutState\_a \subset NAME$ codant l'ensemble des noms des places de l'ASTD **a** est représenté par une fonction totale de $1..card(NAME)$ dans les booléens. Chaque image d'un entier correspondra au booléen qui indique si l'élément appartient à $AutState\_a$.

Si nous prenons comme exemple l'ASTD de la Figure 4.5, l'ensemble $NAME$ de la traduction en B de cet ASTD est $NAME = \{E0, E1F0, E2F0, F1\}$. Dans l'implémentation, cet ensemble devient l'ensemble 1..4 et chaque entier est associé à un état abstrait par exemple $E0 \mapsto 1$, $E1F0 \mapsto 2$, $E2F0 \mapsto 3$, $F1 \mapsto 4$. Enfin, dans notre cas $AutState\_G = NAME$. Ainsi, l'implémentation de $AutState\_G$ est une fonction totale de 1..4 dans l'ensemble des booléens telle que chaque entier est associé à *true*.

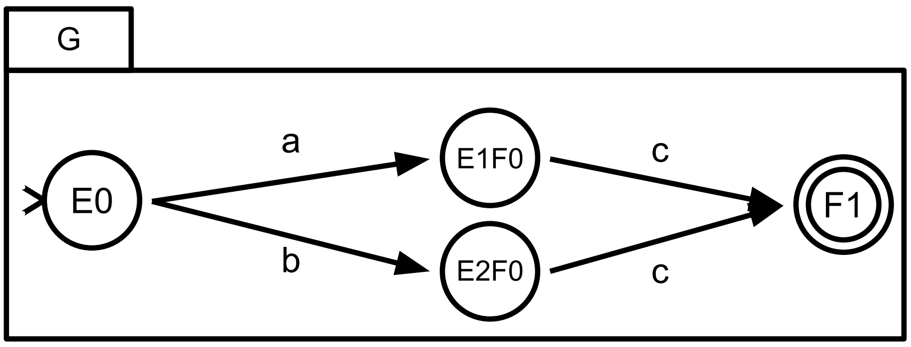


FIGURE 4.5 – Exemple d'ASTD de type automate

Type d'ASTD	Ensemble B			
Sequence	$SEQUENCE$	$fst \mapsto 1$	$snd \mapsto 2$	
Choice	$CHOICE$	$leftS \mapsto 1$	$rightS \mapsto 2$	$none \mapsto 0$
Closure	$KLEENE$	$started \mapsto true$	$notstarted \mapsto false$	
QChoice	$QCHOICE$	$chosen \mapsto true$	$notchosen \mapsto false$	
Guard	$GUARD$	$executed \mapsto true$	$notexecuted \mapsto false$	
Appel d'ASTD	$CALL$	$called \mapsto true$	$notcalled \mapsto false$	

TABLE 4.1 – Implémentation des ensembles spécifiques aux types d'ASTD

Les ensembles spécifiques aux types ASTD Certains ASTD ont besoin d'ensembles spécifiques qui servent à coder l'état de chaque type d'ASTD. Nous détaillons leur implémentation dans la Tableau 4.1. Ce tableau présente pour chaque type d'ASTD l'ensemble B utilisé pour la modélisation de son état. Puis la représentation de chaque élément de l'ensemble abstrait est donnée. Par exemple, l'ensemble $SEQUENCE$ utilisé pour coder l'état d'un ASTD de type séquence est représenté en utilisant l'ensemble 1..2, 1 correspond à l'implémentation de l'état fst et 2 celle de snd.

Les ensembles de quantification Concernant les ensembles de quantification, chaque élément de l'ensemble est associé à un entier, par exemple l'ensemble de quantification T est représenté par $1..card(T)$.

Les fonctions de transition Les fonctions de transition sont modélisées par des fonctions totales retournant un booléen. Les entrées de la fonction sont l'état d'origine et l'état de destination. Le booléen indique si la transition est définie ou non. Si nous appliquons l'implémentation sur l'exemple présenté en Figure 4.5, voici la traduction B des fonctions de transitions de l'automate : $t\_G\_a = \{E0 \mapsto E1F0\}$, $t\_G\_b = \{E0 \mapsto E2F0\}$, $t\_G\_c = \{E1F0 \mapsto F1,\ E2F0 \mapsto F1\}$. L'implémentation de ces fonctions de transition est la représentation sous forme de tableau de fonctions totales suivantes, compte tenu de l'implémentation des places des automates définie précédemment :

$$t\_G\_a\_imp \;=\; \lambda\, x, y\,.\,(x \in 1..4,\ y \in 1..4 \mid bool(x \mapsto y \in \{1 \mapsto 2\}))$$

Cette fonction retourne $true$ lorsque le couple $(1 \mapsto 2)$ est passé en argument, $false$ sinon.

$$\begin{aligned} t\_G\_b\_imp &= \lambda\, x, y\,.\,(x \in 1..4,\ y \in 1..4 \mid bool(x \mapsto y \in \{1 \mapsto 3\})) \\ t\_G\_c\_imp &= \lambda\, x, y\,.\,(x \in 1..4,\ y \in 1..4 \mid bool(x \mapsto y \in \{2 \mapsto 4,\ 3 \mapsto 4\})) \end{aligned}$$

Opérations

Pour chaque type de substitution utilisé dans la traduction, nous présentons leur raffinement. Ce raffinement n'est présenté que quand la substitution n'est pas une substitution d'implémentation. Les substitutions de type **IF** et les affectations (notées :=) sont utilisables directement dans une implémentation.

PRE C THEN S Une substitution précondition n'est pas autorisée dans l'implémentation. Lors du raffinement des machines B, les préconditions doivent être levées.

S1 || S2 Dans la traduction des ASTD en B, les ensembles de variables utilisées par $S1$ et $S2$ sont disjoints. Donc dans notre cas, une substitution simultanée est implémentée en substitution de type séquence. Deux raffinements sont possibles, selon l'ordre que l'on veut appliquer. Le premier opérande en premier ou en second. Nous choisissons d'appliquer la première substitution en premier. Ainsi $S1 \parallel S2$ devient $S1 \,;\, S2$.

SELECT Une substitution sélection s'écrit :

SELECT $C1$ **THEN** $S1$
WHEN $C2$ **THEN** $S2$
...
WHEN Cn **THEN** Sn **END**

Elle est raffinée en substitution de type **CASE** si l'ensemble des $C1 \dots Cn$ sont disjoints. Dans le cas contraire, deux conditions au moins peuvent être vraies simultanément. À l'implémentation, il faut donc choisir l'une des d'appliquer une des substitutions correspondant à une des conditions qui sont vraies. Nous proposons implémenter une substitution **SELECT** en enchainant des substitutions **IF THEN ELSIF**. Ainsi **SELECT C1 THEN S1 WHEN C2 THEN S2 END** devient **IF C1 THEN S1 ELSIF C2 THEN S2 END**. Si plusieurs conditions du **SELECT** sont vraies simultanément, alors seule la substitution correspondant à la première condition vérifiée sera exécutée. Le choix de la substitution à exécuter peut également être laissé au concepteur.

ANY x WHERE P IN S Nous savons que la substitution **ANY** n'est utilisée que dans la traduction des synchronisations quantifiées. Dans notre cas, P est un prédicat de typage de x (de la forme $x \in T\_synch$) en conjonction éventuellement avec une condition sur la valeur de x. De plus, nous avons raffiné les ensembles de quantification en des ensembles d'entiers. Nous pouvons donc itérer sur l'ensemble T_synch des valeurs possibles de la variable quantifiée x. Une substitution de type choix non borné est alors dans notre cas implémentée par une substitution de type boucle. Nous itérons ainsi sur les éléments de l'ensemble des valeurs possibles de la variable de la quantification et dès que l'on trouve un élément satisfaisant P, on applique S pour cette valeur. Cette substitution utilise donc un variant qui contiendra la taille de l'ensemble restant à parcourir.

Exemple d'implémentation

La Figure 4.6 présente un ASTD de type synchronisation quantifiée composé d'un ASTD de type automate. $e1(x)$ représente un évènement non synchronisé dont la variable de quantification apparait dans les paramètres. $e2$ est un évènement synchronisé. $e3$ est un évènement non synchronisé qui n'utilise pas la variable de quantification. La traduction en B de l'évènement $e3$ de l'ASTD présenté dans la Figure 4.6 contient donc une substitution **ANY**.

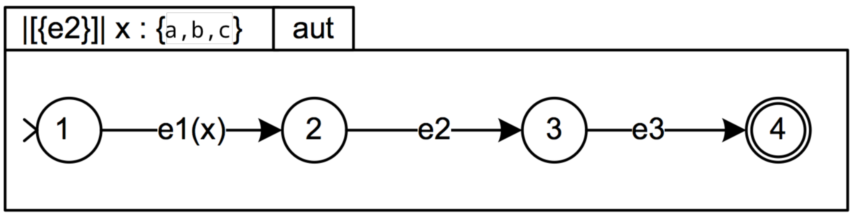


FIGURE 4.6 – Un ASTD de type synchronisation quantifiée

```
SETS
    AutState_aut = {s1, s2, s3, s4} ;
    T_synch = {aa, bb, cc}
    /* l'usage de termes B comportant un seul caractère n'est
    pas autorisé par Atelier B. Ainsi 'aa' représente 'a'.*/

INVARIANT
    State_aut : T_synch --> AutState_aut

OPERATION
e3 =
PRE
    ∃ vv.( vv ∈ T_synch ∧ State_aut(vv) ∈ dom ( t_aut_e3 ))
THEN
    ANY vv WHERE vv ∈ T_synch ∧ State_aut(vv) ∈ dom ( t_aut_e3 ) THEN
        State_aut(vv) := t_aut_e3 ( State_aut(vv) )
    END
END
```

L'évènement *e*3 est implémenté de la manière présenté suivante :

```
CONSTANTS
    C_AutState_aut = 0..3 ;
    C_T_synch = 0..2

OPERATION
e3 =
    VAR compteur IN
        compteur := 0 ;
        WHILE (compteur < 3 )
        DO
            IF C_State_aut(compteur) ∈ dom ( C_t_aut_e3 )
            THEN
                C_State_aut(compteur) := C_t_aut_e3 ( C_State_aut(compteur) ) ;
                compteur := 3
            ELSE
                compteur := compteur + 1
            END
        INVARIANT
            compteur : 0..3
        VARIANT
            3 - (compteur)
        END
    END
```

Nous avons implémenté l'ensemble de quantification $T\_synch = \{aa, bb, cc\}$ par l'ensemble $C\_T\_synch = 0..2$. La valeur 0 correspond à aa, 1 a bb et 2 à cc. De même, $AutState\_aut = \{s1, s2, s3, s4\}$ est implémenté par $C\_AutState\_aut = 0..3$. Enfin, la substitution **ANY** est implémentée par une boucle basée sur une variable nommée *compteur*. Cette variable représente la valeur courante de C_T_synch qui est testée pour vérifier que le prédicat du **ANY** est vrai. Ce prédicat se retrouve dans la condition du **IF**. La substitution du **ANY** n'est appliquée que quand ce prédicat est vrai, et la variable compteur prend alors une valeur qui permet d'interrompre la boucle. La précondition de l'opération $e3$ abstraite garantit qu'une valeur du compteur permettra d'appliquer la substitution.

4.4 Vers une traduction BPEL

Dans le cadre de leurs travaux, Embe-Jiague *et al.* ont proposé dans [7] une traduction BPEL des ASTD. BPEL (*Business Process Execution Language*) [32] est un langage d'expression de processus métiers utilisant la syntaxe de XML. Il permet de définir des activités, la façon dont elles s'enchainent, le tout dans le cadre d'une architecture orientée services. Les activités BPEL sont décomposées en deux types. Les activités basiques décrivent les étapes élémentaires du processus. Un exemple d'activité basique est l'affection d'une valeur à une variable. Les activités structurées permettent de composer plusieurs activités ensemble. Elles sont en ce sens similaires aux opérateurs des algèbres de processus. Les types des variables sont ceux disponibles avec XML notamment les types numériques et les chaines de caractères, les types énumérés, les listes etc.

Dans son implémentation, une machine B modélisant le comportement dynamique d'une politique de contrôle d'accès ne comporte plus que certains types de substitutions. Comme détaillé dans la section précédente, les seules substitutions disponibles dans une implémentation sont : l'affectation ; la séquence ; la définition d'une variable locale ; la structure conditionnelle **IF** ; la boucle **WHILE** et les appels d'opérations externes.

Nous pensons que ces substitutions peuvent être traduites assez intuitivement en activités BPEL. Nous n'avons cependant pas formellement défini de règles de traduction mais les types de données et les substitutions des implémentations B obtenues nous laissent penser que cette transformation est possible.

– L'affectation B peut être traduite par une balise `<assign>` modélisant une activité d'assignation (*assignment*) en BPEL.
– La séquence B peut être traduite par une balise `<sequence>` modélisant une activité de type séquence en BPEL.
– La définition d'une variable locale B peut être traduite par une balise `<variable>` pour définir une variable dont la portée peut être limitée par un `<scope>`.
– Le **IF THEN ELSE** en B peut être traduit par les balises `<if>` `<then>` `<elseif>` `<else>` en BPEL. Les conditions sont alors exprimées dans la balise `<condition>`.
– La boucle **WHILE** en B peut être traduite par une balise `<while>` dans laquelle on ajoute également une balise `<condition>`.

Concernant les types de données des variables de l'implémentation B, les types booléens, entiers, tableaux d'entiers, tableaux de booléens peuvent être codés en XML et sont donc définis en BPEL. Nous ne voyons donc pas, *a priori*, de restriction pour la traduction des implémentations B modélisant l'aspect dynamique d'une spécification de contrôle d'accès en BPEL. Cette traduction nous permettrait de comparer les différentes implémentations des politiques de contrôle d'accès exprimées à l'aide de la notation ASTD.

4.5 Conclusion

Dans ce chapitre, nous avons fait le lien entre les classes du méta-modèle MMB_{sec} présenté dans le chapitre précédent et le méta-modèle MM_{imp} introduit par Embe Jiague *et al.* dans [8]. Nous avons également décrit le diagramme de séquence décrivant le processus d'interrogation du filtre de contrôle d'accès lors d'une requête de l'utilisateur. Nous avons également présenté la façon dont sont implémentées les machines B obtenues par la traduction de spécifications ASTD ainsi que le filtre de contrôle d'accès. Puisque ces machines utilisent seulement un sous-ensemble des types B disponibles et que leur structure est connue d'avance, un raffinement automatique vers une implémentation peut être envisagé. De plus, une étude rapide du langage BPEL montre qu'une traduction vers BPEL depuis l'implémentation B semble possible. Il faudra tout de même valider cette hypothèse si l'on souhaite utiliser cette approche. Il sera alors intéressant de comparer la traduction d'un ASTD vers BPEL avec la traduction passant par B et son raffinement avant d'être traduite en BPEL.

Quatrième partie

Conclusion

Nous avons présenté deux approches de traduction PIM-PSM dans le cadre de la modélisation de politiques de contrôle d'accès. La première est une traduction d'un PIM ASTD modélisant les règles statiques et dynamiques de contrôle d'accès. La seconde est une traduction en B d'un PIM comportant trois modèles graphiques ASTD, SecureUML et UML représentant respectivement les aspects dynamiques, statiques et fonctionnels d'un SI. Cette spécification B est ensuite raffiné pour obtenir un PSM en B n'utilisant qu'un sous-ensemble directement implémentable de B. Ce modèle peut alors être implémenté automatique par le biais d'une traduction systématique.

Si nous comparons les deux approches, la première ne prend pas en compte les aspects fonctionnels du système. Ce point rend impossible la vérification de la cohérence de la politique de contrôle d'accès avec le système. Dans la seconde approche, le modèle B combine tous les aspects de la politique et du SI ce qui permet d'animer et de prouver des propriétés de sécurité sur le couple SI/Politique de contrôle d'accès. De plus, les étapes de traduction et de raffinement de la seconde approche peuvent être prouvées ce qui garantit que l'implémentation obtenue en PSM correspond bien au PIM. Ce genre de preuve ainsi que la vérification de propriétés de sécurité justifient l'utilisation des méthodes formelles dans le cadre de la spécification de politiques de contrôle d'accès.

Bibliographie

[1] Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for Web services, 2003.

[3] Nazim Benaissa. *La composition des protocoles de sécurité avec la méthode B événementielle.* These, Université Henri Poincaré - Nancy I, May 2010.

[4] The ORKA Consortium. Orka — organizational control architecture, 2010.

[5] Frédéric Cuppens and Alexandre Miège. Modelling contexts in the Or-BAC model. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, pages 416–, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. Organization based access control. In *POLICY '03 : Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 120, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Michel Embe Jiague, Marc Frappier, Frédéric Gervais, Régine Laleau, and Richard St-Denis. Enforcing ASTD access control policies to WS-BPEL processes deployed in a SOA environment. *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, 2(2) :37–59, 2011.

[8] Michel Embe Jiague, Marc Frappier, Frédéric Gervais, Régine Laleau, and Richard St-Denis. A metamodel for the design of access-control policy enforcement managers. In *Foundations & Practice Of Security (FPS 2011)*, volume 6888 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011.

[9] Michel Embe-Jiague, Richard St-Denis, Régine Laleau, and Frédéric Gervais. A bpel implementation of a security filter. In *PhD Symposium of 8th European Conference on Web Services*, 2010.

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

[11] Benoît Fraikin, Frédéric Gervais, Marc Frappier, Régine Laleau, and Mario Richard. Synthesizing information systems : the APIS project. In Colette Rolland, Oscar Pastor, and Jean-Louis Cavarero, editors, *First International Conference on Research Challenges in Information Science (RCIS)*, pages 73–84, Ouarzazate, Morocco, April 2007.

[12] M. Frappier, F. Diagne, and A. Amel Mammar. Proving reachability in B using substitution refinement. In Elsevier, editor, *B 2011 Workshop*, volume to appear of *Electronic Notes in Theoretical Computer Science*, 2011.

[13] Marc Frappier, Frédéric Gervais, Régine Laleau, Benoît Fraikin, and Richard St-Denis. Extending statecharts with process algebra operators. *Innovations in Systems and Software Engineering*, 4(3) :285–292, October 2008.

[14] Marc Frappier and Richard St-Denis. EB^3 : an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2 :134–149, 2003.

[15] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. Informal and formal requirements specification laguages : Bridging the gap. *IEEE Transactions on Software Engineering*, 17(5) :454–465, 1991.

[16] D. Harel. Statecharts : A visual formalism for complex systems. *Science of computer programming*, 8(3) :231–274, 1987.

[17] Akram Idani. *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B.* Thèse, Université Joseph-Fourier - Grenoble I, 2006.

[18] Akram Idani, Mohamed-Amine Labiadh, and Yves Ledru. Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *Ingénierie des Systèmes d'Information*, 15(3) :87–112, 2010.

[19] Jan Jürjens. Umlsec : Extending uml for secure systems development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *«UML» 2002 — The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg, 2002.

[20] Regine Laleau and Amel Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 269–272, Washington, DC, USA, 2000. IEEE Computer Society.

[21] Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B : Formal Verification of Object-Oriented Models. In *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 187–206. Springer, 2004.

[22] Yves Ledru, Akram Idani, Jérémy Milhau, Nafees Qamar, Régine Laleau, Jean-Luc Richier, and Mohamed-Amine Labiadh. Taking into account functional models in the validation of is security policies. In Camille Salinesi, Oscar Pastor, Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, and Clemens Szyperski, editors, *Advanced Information Systems Engineering Workshops*, volume 83 of *Lecture Notes in Business Information Processing*, pages 592–606. Springer Berlin Heidelberg, 2011.

[23] Michael Leuschel and Michael Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Berlin / Heidelberg, 2003.

[24] Michael Leuschel and Michael J. Butler. ProB : an automated analysis toolset for the B method. *STTT*, 10(2) :185–203, 2008.

[25] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml : A uml-based modeling language for model-driven security. In *5th International Conference on The Unified Modeling Language (UML)*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.

[26] Amel Mammar, Marc Frappier, and Fama Diagne. A proof-based approach to verifying reachability properties. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1651–1657, New York, NY, USA, 2011. ACM.

[27] Amel Mammar and Régine Laleau. Implémentation JAVA d'une spécification B : Application aux bases de données. *Technique et Science Informatiques*, 27(5) :537–570, 2008.

[28] Jérémy Milhau, Marc Frappier, Frédéric Gervais, and Régine Laleau. Systematic translation of EB3 and ASTD specifications in B and EventB. Technical Report 30 v3.0, Université de Sherbrooke, 2010.

[29] Jérémy Milhau, Marc Frappier, Frédéric Gervais, and Régine Laleau. Systematic translation rules from ASTD to Event-B. In Dominique Méry and Stephan Merz, editors, *Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin / Heidelberg, 2010.

[30] Jérémy Milhau, Akram Idani, Régine Laleau, Mohamed Amine Labiadh, Yves Ledru, and Marc Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. *In International NASA Journal on Innovations in Systems and Software Engineering (ISSE), Special Issue of UMLFM 2011 workshop*, to be published, 2011.

[31] Gustaf Neumann and Mark Strembeck. An approach to engineer and enforce context constraints in an rbac environment. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 65–79, New York, NY, USA, 2003. ACM.

[32] OASIS. *Web Services Business Process Execution Language Version 2.0*. OASIS, 2007.

[33] Stere Preda, Nora Cuppens-Boulahia, Frédéric Cuppens, Joaquin Garcia-Alfaro, and Laurent Toutain. Model-driven security policy deployment : Property oriented approach. In *International Symposium on Engineering Secure Software and Systems (ESSOS'10)*, volume 5965 of *LNCS*, pages 123–139. Springer, 2010.

[34] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2) :38–47, 1996.

[35] Colin Snook and Michael Butler. U2B - A tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, 2004.

[36] Colin Snook and Michael Butler. UML-B : Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :92–122, 2006.

[37] Manachai Toahchoodee, Indrakshi Ray, Kyriakos Anastasakis, Geri Georg, and Behzad Bordbar. Ensuring spatio-temporal access control for real-world applications. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, pages 13–22, New York, NY, USA, 2009. ACM.