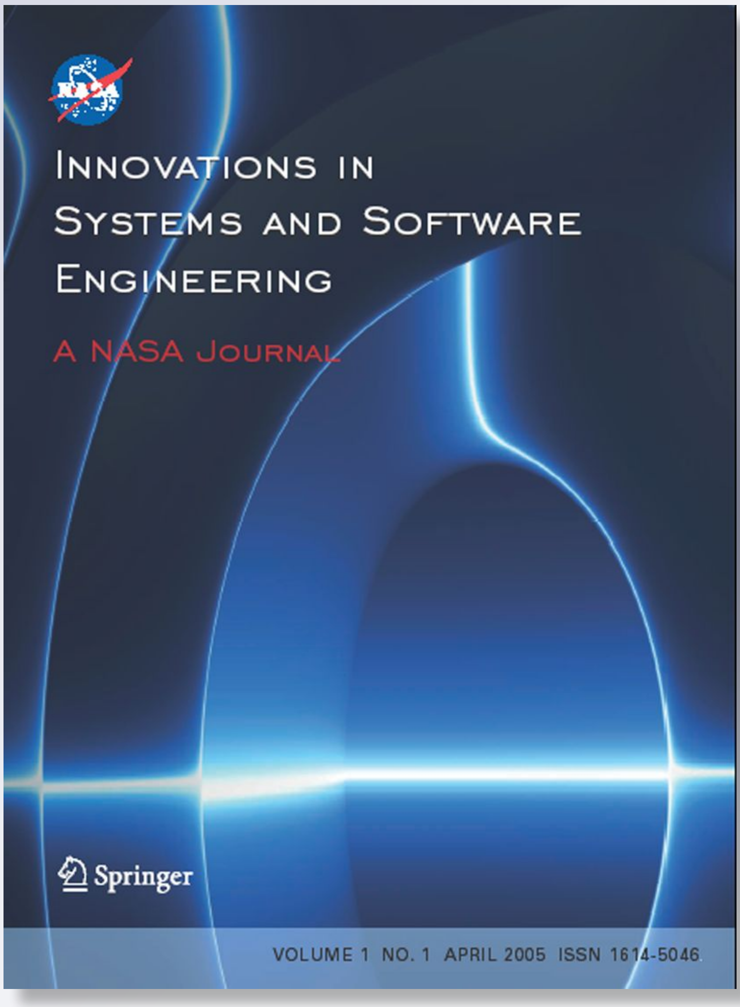
Combining UML, ASTD and B for the formal specification of an access control filter

J. Milhau, A. Idani, R. Laleau, M. A. Labiadh, Y. Ledru & M. Frappier

**Innovations in Systems and Software Engineering**
A NASA Journal

ISSN 1614-5046

Innovations Syst Softw Eng
DOI 10.1007/s11334-011-0166-z


Innovations in Systems and Software Engineering
A NASA Journal
Springer
VOLUME 1 NO. 1 APRIL 2005 ISSN 1614-5046

Springer

Your article is protected by copyright and all rights are held exclusively by Springer-Verlag London Limited. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your work, please use the accepted author's version for posting to your own website or your institution's repository. You may further deposit the accepted author's version on a funder's repository at a funder's request, provided it is not made publicly available until 12 months after publication.

Springer

Author's personal copy

Innovations Syst Softw Eng
DOI 10.1007/s11334-011-0166-z

SI : FM & UML

# Combining UML, ASTD and B for the formal specification of an access control filter

**J. Milhau · A. Idani · R. Laleau · M. A. Labiadh · Y. Ledru · M. Frappier**

Received: 30 April 2011 / Accepted: 1 September 2011
© Springer-Verlag London Limited 2011

**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

J. Milhau (✉) · M. Frappier
GRIL, Département Informatique, Université de Sherbrooke,
2500 boulevard université, Sherbrooke, QC J1K 2R1, Canada
e-mail: Jeremy.Milhau@USherbrooke.ca

M. Frappier
e-mail: Marc.Frappier@USherbrooke.ca

J. Milhau · R. Laleau
Département Informatique, LACL, IUT Sénart Fontainebleau,
Université Paris-Est, Route Hurtault, 77300 Fontainebleau, France
e-mail: Laleau@u-pec.fr

A. Idani · M. A. Labiadh · Y. Ledru
Laboratoire d'Informatique de Grenoble UMR 5217,
UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble2/CNRS,
38041 Grenoble, France
e-mail: Akram.Idani@imag.fr

M. A. Labiadh
e-mail: Mohamed-Amine.Labiadh@imag.fr

Y. Ledru
e-mail: Yves.Ledru@imag.fr

## 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 [6]. 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 [19] to express static rules and the ASTD notation [8] for dynamic

Published online: 29 September 2011

Author's personal copy

rules. These notations are presented in Sect. 3. These graphical specifications are then translated into B machines [1] using translation rules described in Sect. 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 Sect. 5. We start in the next section by describing the context of the work.

## 2 Context

### 2.1 The Selkis project

The Selkis project[1] funded by the French national research agency (ANR) aims to define a development strategy for a 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 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. 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,

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

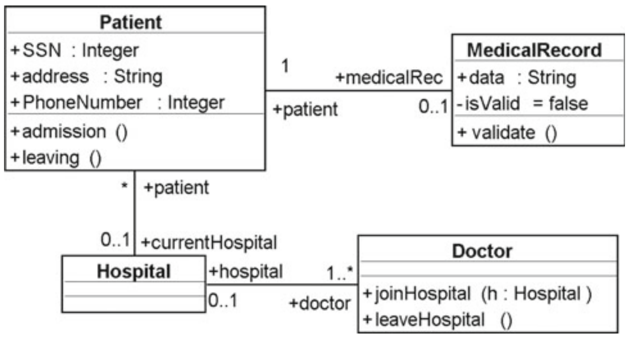


**Fig. 1** Functional model

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

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.

## 3 Graphical models for access control

In our methodology we propose to use SecureUML [19] to model static access control, and ASTD [8] diagrams for dynamic access control. Static access control is based on RBAC (Role-Based Access Control) [24], which describes authorizations granted to users on resources. Dynamic access control rules can refer to previous states of the IS.

### 3.1 SecureUML

SecureUML [19] 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

Author's personal copy

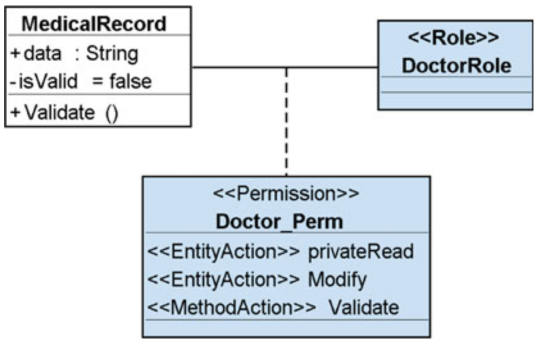


**Fig. 2** Access control rules for medical records

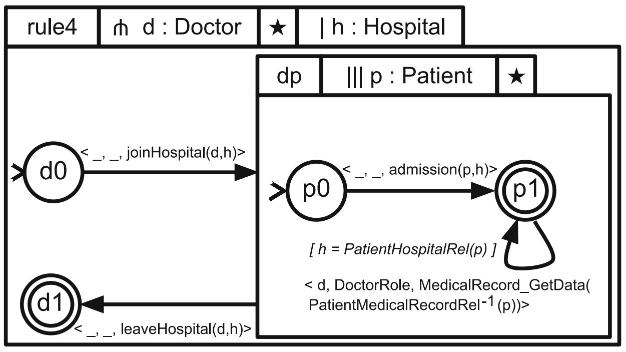

rule4
⫛ d : Doctor
★
| h : Hospital
dp
||| p : Patient
★
d0
<_, _, joinHospital(d,h)>
p0
<_, _, admission(p,h)>
p1
[ h = PatientHospitalRel(p) ]
< d, DoctorRole, MedicalRecord_GetData( PatientMedicalRecordRel-1 (p))>
d1
<_, _, leaveHospital(d,h)>

**Fig. 3** ASTD model of rule R4: If a patient has left the hospital, only doctors belonging to the hospital during the patients stay will keep read access to his medical record

order to model roles and their permissions. This is the main reason why we have chosen SecureUML rather than UMLSec [13]. In SecureUML, users are grouped into roles and may play several roles with respect to the secure system. Figure 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≫ 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 readable, rule R3 is described using a B predicate in Sect. 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 that explicitly models states and transitions between states is more intuitive than manually coding 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.

## 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 [11] and is based on operators from $\text{EB}^3$ [9] (a process algebra dedicated to IS specifications). Readers are invited to consult a formal and mathematical description of the ASTD notation in [8].

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. 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 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

Author's personal copy

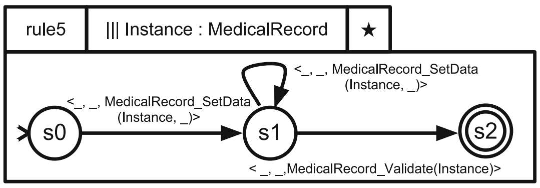

rule5
||| Instance : MedicalRecord
★
<_, _, MedicalRecord_SetData (Instance, _)>
<_, _, MedicalRecord_SetData (Instance, _)>
s0
s1
s2
<_, _,MedicalRecord_Validate(Instance)>

**Fig. 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

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

$$< d, DoctorRole, \text{MedicalRecord\_GetData}(\ i\ ) >$$

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

In OrBAC [4], 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 [3] 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.

## 4 Translations into B specifications

The idea of integrating formal and graphical notations (i.e. B and UML) has been studied since several years [10], 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.

### 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

Author's personal copy

steps of reformulation are called refinements and the last one is the implementation.

## 4.2 A formal functional model

The initial functional model (Fig. 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 [14,15,25]. In order to take advantage of these complementary works we integrated their translation rules in a unified MDE framework [12]. 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 [26] 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 [26] 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. 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*. Figure 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. 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:

```
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;
```

**Fig. 5** Basic B structures related to medical records

```
result, executed ← MedicalRecord__GetData(Instance) ≙
    PRE
        Instance ∈ MedicalRecord
    THEN
        result := MedicalRecord__data(Instance) ||
        executed := ok
    END;
```

**Fig. 6** Operation *MedicalRecord__GetData*

$$\forall\, pp \cdot (pp \in Patient \wedge \\ MedicalRecord\_\_isValid(PatientMedicalRecordRel^{-1}(pp)) = \mathbf{FALSE} \Rightarrow \\ PatientHospitalRel[\{pp\}] \neq \emptyset\,)$$

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*. Figure 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.

## 4.3 A formal static access control model

To our knowledge there is no attempt to translate SecureUML models into B. Nevertheless, in [23] the authors gave

Author's personal copy

```
executed ← MedicalRecord__SetData(Instance, data) ≙
    PRE
        Instance ∈ MedicalRecord ∧
        data ∈ THEDATA
    THEN
        IF
            PatientMedicalRecordRel(Instance) ∈
                                dom(PatientHospitalRel)
        THEN
            MedicalRecord__data(Instance) := data ||
            MedicalRecord__isValid(Instance) := FALSE ||
            executed := ok
        ELSE
            executed := failure
        END
    END;
```

**Fig. 7** Operation *MedicalRecord__SetData*

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

**Fig. 8** Operation *secure_MedicalRecord__SetData*

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.

```
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:

$$\begin{array}{l}\mathrm{PermissionAssignement} \in \mathit{PERMISSIONS} \rightarrow (\mathit{ROLES} \times \mathit{ENTITIES})\\ \wedge\ \mathit{Roles\_Hierarchy} \in \mathit{ROLES} \longleftrightarrow \mathit{ROLES}\ \wedge\\ \mathbf{closure1}(\mathit{Roles\_Hierarchy}) \cap \mathbf{id}(\mathit{ROLES}) = \emptyset\end{array}$$

This means that a permission links at the most one pair (role ↦ 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 initialization 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 initialization clause is given below:

```
INITIALISATION
AttributeOf := {(data ↦ MedicalRecord), ...}
AttributeKind := {(data ↦ public), ...}
OperationOf := {(MedicalRecord__SetData ↦ MedicalRecord), ...}
setterOf := {(MedicalRecord__SetData ↦ data), ...}
PermissionAssignement :=
      {(Doctor_Perm ↦ (DoctorRole ↦ MedicalRecord)), ...}
EntityActions := {(Doctor_Perm ↦ {privateRead, modify}), ...}
...
```

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. 8 is intended to verify accesses to the functional operation *MedicalRecord__SetData* of Fig. 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* ∈ *roleOf*(*user*)). Predicate "*MedicalRecord__SetData* ∈ *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* ↦ *MedicalRecord__SetData*).

As mentioned in Sect. 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`).

Author's personal copy

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

**Fig. 9** Operation *secure_MedicalRecord__GetData*

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*) ≙
$(user \in Doctor \Rightarrow HospitalDoctorRel(user) = PatientHospitalRel(PatientMedicalRecordRel(instance)))$

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 [16].

Figure 9 details the operation used in the rest of this paper: *secure_MedicalRecord__GetData*. This operation is another example of the translation of the SecureUML model into B.

## 4.4 Dynamic access control model translation

In [22] 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 [22] for B as described in [21]. 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.

```
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
```

**Fig. 10** Operation *Dynamic_MedicalRecord__GetData*

Figure 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. 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. 3; it ensures that 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 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 [21].

Author's personal copy

## 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 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.

### 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. 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. 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.

```
Rollback() ≙
    PRE
        rollbacklock = locked
    THEN
        rollbacklock := unlocked
    END

answer, executed, data
  ← Filter_MedicalRecord__GetData(Instance, user, role) ≙
    PRE
        Instance ∈ MedicalRecord ∧
        user ∈ USERS ∧
        role ∈ ROLES ∧
        rollbacklock = unlocked
    THEN
        CHOICE
            answer := granted || executed := ok
            || data :∈ THEDATA || rollbacklock := unlocked
          OR
            answer := granted || executed := notExecuted
            || data :∈ THEDATA || rollbacklock := locked
          OR
            answer := denied || executed :∈ {ok, notExecuted}
            || data :∈ THEDATA
            || rollbacklock :∈ {locked, unlocked}
        END
    END
```

**Fig. 11** Abstract operations *Rollback* and *Filter_MedicalRecord__GetData*

### 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. Figure 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. 12 introduces the abstract access control policy specification composed by all the translated B machines.

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. Figure 13 is the refinement of operation *Filter_MedicalRecord__GetData* introduced in Fig. 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

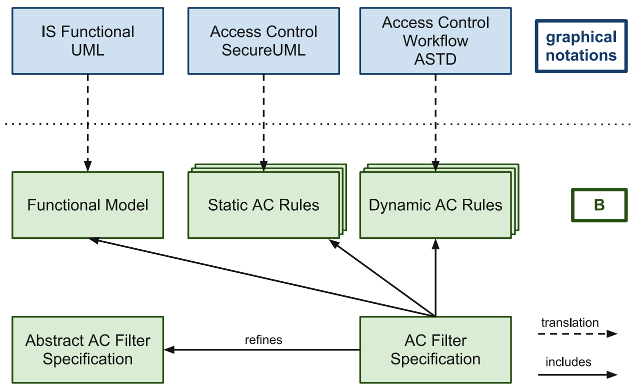


**Fig. 12** Overall view of the access control filter

Author's personal copy

```
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
```

**Fig. 13** Refinement of *Filter_MedicalRecord__GetData*

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 [5]. When IS are implemented using Web services, 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.

## 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 [17] 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 [17] or using the proof-based verification approach developed by Mammar et al. [20] and Frappier et al. [7]. For instance, we could verify that rule R5 is correctly enforced by checking that the CTL property

$$\mathbf{AG}(\neg\, MedicalRecord\_isValid(i) \Rightarrow$$
$$\mathbf{AGEF}\, pre(MedicalRecord\_Validate(i)))$$

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.

Author's personal copy

## 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 [17,18]. 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 [4] 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 [19], 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 [27], where functional and security models are merged into a single UML model which is translated into Alloy. However, the access control rules described in [27] 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.

**Acknowledgments** This research is funded by ANR (France) as part of the Selkis project (ANR-08-SEGI-018) and by NSERC (Canada). We would like to thank the anonymous referees for their useful comments and suggestions on this work.

## References

1. Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press
2. Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, Liu K, Roller D, Smith D, Thatte S, Trickovic I, Weerawarana S (2003) Business process execution language for Web services
3. Cuppens F, Miège A (2003) Modelling contexts in the Or-BAC model. In: Proceedings of the 19th annual computer security applications conference, ACSAC '03. IEEE Computer Society, Washington, p 416
4. El Kalam AA, Benferhat S, Miège A, El Baida R, Cuppens F, Saurel C, Balbiani P, Deswarte Y, Trouessin G (2003) Organization based access control. In: POLICY '03: proceedings of the 4th IEEE international workshop on policies for distributed systems and networks. IEEE Computer Society, Washington, p 120
5. Embe Jiague M, Frappier M., Gervais F, Laleau R, St-Denis R (2011) Enforcing ASTD access control policies to WS-BPEL processes deployed in a SOA environment. Int J Syst Service-Oriented Eng 2(2):37–59
6. Ferraiolo DF, Kuhn DR, Chandramouli R (2003) Role-based access control. Artech House Inc., Norwood
7. Frappier M, Diagne F, Amel Mammar A (2011) Proving reachability in B using substitution refinement. In: B 2011 Workshop. Electronic Notes in Theoretical Computer Science (to appear)
8. Frappier M, Gervais F, Laleau R, Fraikin B, St-Denis R (2008) Extending statecharts with process algebra operators. Innov Syst Softw Eng 4(3):285–292
9. Frappier M, St-Denis R (2003) EB$^3$: an entity-based black-box specification method for information systems. Softw Syst Model 2:134–149
10. Fraser MD, Kumar K, Vaishnavi VK (1991) Informal and formal requirements specification languages: bridging the gap. IEEE Trans Softw Eng 17(5):454–465
11. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274
12. Idani A, Labiadh MA, Ledru Y (2010) Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. Ingénierie des Systèmes d'Information 15(3):87–112
13. Jürjens J (2002) Umlsec: extending UML for secure systems development. In: Jézéquel JM, Hussmann H, Cook S (eds) UML 2002—the unified modeling language. Lecture notes in computer science, vol 2460. Springer, Heidelberg, pp 1–9
14. Laleau R, Mammar A (2000) 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. IEEE Computer Society, Washington, pp 269–272
15. Lano K, Clark D, Androutsopoulos K (2004) UML to B: formal verification of object-oriented models. In: Integrated formal methods. Lecture notes in computer science, vol 2999. Springer, pp 187–206
16. Ledru Y, Idani A, Milhau J, Qamar N, Laleau R, Richier JL, Labiadh MA (2011) Taking into account functional models in the validation of is security policies. In: Salinesi C, Pastor Q, Aalst W, Mylopoulos J, Sadeh NM, Shaw MJ, Szyperski C (eds) Advanced information systems engineering workshops. Lecture notes in business information processing, vol 83. Springer, Heidelberg, pp 592–606
17. Leuschel M, Butler M (2003) ProB: a model checker for B. In: Araki K, Gnesi S, Mandrioli D (eds) FME 2003: formal methods. Lecture notes in computer science, vol 2805. Springer, Heidelberg pp 855–874
18. Leuschel M, Butler MJ (2008) ProB: an automated analysis toolset for the B method. STTT 10(2), 185–203
19. Lodderstedt T, Basin DA, Doser J (2002) Secureuml: a UML-based modeling language for model-driven security. In: 5th International conference on the unified modeling language (UML). LNCS, vol 2460. Springer, Berlin, pp 426–441
20. Mammar A, Frappier M, Diagne F (2011) A proof-based approach to verifying reachability properties. In: Proceedings of the 2011 ACM symposium on applied computing, SAC '11. ACM, New York, pp 1651–1657
21. Milhau J, Frappier M, Gervais F, Laleau R (2010) Systematic translation of EB3 and ASTD specifications in B and EventB. Technical report 30 v3.0, Université de Sherbrooke
22. Milhau J, Frappier M, Gervais F, Laleau R (2010) Systematic translation rules from ASTD to Event-B. In: Méry D, Merz S (eds)

Author's personal copy

Integrated formal methods. Lecture notes in computer science, vol 6396. Springer, Berlin, pp 245–259

23. Preda S, Cuppens-Boulahia N, Cuppens F, Garcia-Alfaro J, Toutain L (2010) Model-driven security policy deployment: property oriented approach. In: International symposium on engineering secure software and systems (ESSOS'10). LNCS, vol 5965. Springer, Berlin, pp 123–139
24. Sandhu R, Coyne E, Feinstein H, Youman C (1996) Role-based access control models. IEEE Comput 29(2):38–47
25. Snook C, Butler M (2004) U2B-A tool for translating UML-B models into B. In: Mermet J (ed) UML-B specification for proven embedded systems design
26. Snook C, Butler M (2006) UML-B: formal modeling and design aided by UML. ACM Trans Softw Eng Methodol 15(1):92–122
27. Toahchoodee M, Ray I, Anastasakis K, Georg G, Bordbar B (2009) Ensuring spatio-temporal access control for real-world applications. In: Proceedings of the 14th ACM symposium on access control models and technologies, SACMAT '09. ACM, New York, pp 13–22