Taking into Account Functional Models in the Validation of IS Security Policies

Yves Ledru[1], Akram Idani[1], Jérémy Milhau[2,3], Nafees Qamar[1], Régine Laleau[2], Jean-Luc Richier[1], and Mohamed-Amine Labiadh[1,⋆]

[1] UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble2/CNRS, Laboratoire d'Informatique de Grenoble UMR 5217, F-38041, Grenoble, France
Yves.Ledru@imag.fr

[2] Université Paris-Est, LACL, IUT Sénart Fontainebleau, Fontainebleau, France
laleau@u-pec.fr

[3] GRIL, Département Informatique, Université de Sherbrooke, Québec, Canada
Jeremy.Milhau@USherbrooke.ca

Abstract. Designing a security policy for an information system (IS) is a non-trivial task. Variants of the RBAC model can be used to express such policies as access-control rules associated to constraints. In this paper, we advocate that currently available tools do not take sufficiently into account the functional description of the application and its impact on authorisation constraints and dynamic aspects of security. We suggest to translate both security and functional models into a formal language, such as B, whose analysis and animation tools will help validate a larger set of security scenarios. We show how various kinds of constraints can be expressed and animated in this context.

Keywords: RBAC, authorisation constraints, validation.

1 Introduction

The design of today's information systems (IS) must not only take into account the expected functionalities of the system, but also various kinds of non-functional requirements. Security is one of these non-functional requirements. Security policies are designed to fulfill requirements such as confidentiality, integrity and availability. Security policies are usually expressed as abstract access control rules, independently of target technologies. In the past, various access control models have been proposed. In this paper, we focus on role-based access control models (RBAC) [1], including evolutions such as SecureUML [2]. An important feature of such models is the notion of role: permissions are granted to roles which represent functions in an institution. Each role corresponds to several users and users may play several roles with respect to the secure system.

⋆ This work was partly supported by the ANR-08-SEGI-018 Selkis and ANR-09-SEGI-014 TASCCC projects.

C. Salinesi and O. Pastor (Eds.): CAiSE 2011 Workshops, LNBIP 83, pp. 592–606, 2011.

Advanced RBAC models allow to express constraints such as Separation of Duty (SoD) properties [3], and other properties on roles (e.g. precedence, see Sect. 2). For information systems, contextual information may also be taken into account when granting permissions. This contextual information may correspond to the current state of the information system, or to the history of interactions with the system. This reveals the need to link the security model of the application to the functional model of the information system, often expressed as UML diagrams. Therefore, SecureUML [2] groups UML diagrams of the application with security information describing the access control rules. In the remainder, we will refer to the UML diagrams of the application as the *functional model*. The term *security model* will refer to the access control model. Other approaches have integrated security concerns in UML diagrams. Fernandez [4] proposes to address security concerns through the whole software development and builds on UML and patterns to structure his approach. UMLSec [5] is another UML profile that focuses on secrecy and cryptographic protocols.

Contextual constraints give flexibility to describe security policies, but the resulting models are more complex to validate. Validation checks that the policy corresponds to the user's requirements. Animation of the model can play a significant role in this validation activity, playing scenarios or answering questions about the consequences of the model. Animation also brings a limited level of verification: traces demonstrate that constraints are not contradictory.

When systems become complex, separation of concerns is often perceived as a good strategy to master complexity. In our context, this means that functional and security models should be validated separately. This explains why most existing works are mainly interested by the security part. Although it is definitely useful to first analyse both models in isolation, interactions between these models must also be taken into account. Such interactions result from the fact that constraints expressed in the security model also refer to information of the functional model. Hence, evolutions of the functional state will influence the security behaviour. Conversely, security constraints can impact the functional behaviour. For example, it is important to consider both security and functional models in order to check liveness properties on the information system. Indeed, it can be the case that security constraints are too strict and block the system.

In this paper we review several tools aimed at the validation of RBAC security properties, and state that most of them focus on the security model without taking into account the functional model. We will then propose an approach, based on the B method [6], which allows to express contextual and history based constraints, and to validate these models using animation tools.

In Sect. 2, we review several tools representative of the current state of the art. In Sect. 3, we present an example whose validation requires to take into account dynamic aspects of the functional model. Such aspects cannot be investigated with current tools. Sect. 4 proposes solutions based on the B formal method to address these issues. Finally Sect. 5 draws the conclusions of this work.

2 Tools for V&V of Role-Based Authorisation Constraints

2.1 Validation Tools Based on OCL

OCL [7] (Object Constraint Language) is part of UML and allows to express invariant constraints on a class diagram as well as pre- and postconditions on the methods. The USE tool (UML-based Specification Environment) [8] takes as input an object diagram and an OCL constraint. It checks whether the constraint holds on the given object diagram. The tool also allows to program a random generator for object diagrams, and to program sequences of object diagrams, where pre- and post-conditions can be checked.

Sohr et al [9] have adapted this tool for the analysis of security policies. Their work focuses on the security model, i.e. users, roles, sessions and permissions, constrained by OCL assertions. This allows to express properties such as the cardinality of a given role, the precedence between roles (e.g. only members of role r_1 may be assigned role r_2), or SoD (i.e. conflicting roles). Their work also takes into account a limited amount of functional information by adding some attributes to the users. For example, if a constraint states that the doctor accessing medical information about a patient must be linked to the hospital of the patient, some attribute *currentHospital* should be added to the users. Unfortunately such extensions of the security model don't really scale up, and duplicate information already included in the functional model.

Sohr et al [9] report on two kinds of validation activities. An object diagram can be given to the tool, and the tool will check which constraints are violated. The object diagram can be user-defined, randomly generated, or member of a programmed sequence. This allows to detect unsatisfiable constraints, i.e. constraints which are always false. They have also developed a tool named authorisation editor, which implements the administrative, system and review functions of the RBAC standard. The tool is connected to the API of USE so that the constraints of the security policy are checked after each operation. This allows to detect erroneous dynamic behaviour of the security policy. For example, if two roles are constrained both by a precedence and a conflict relations, it will always be impossible to find a sequence of RBAC administrative and system operations which leads to create the second role.

Other works have addressed the validation of security policies using UML and OCL. They focus on SoD properties in the security model. Ahn and Hu [10] stated an approach using UML class diagrams, a language dedicated to the specification of role-based authorization constraints (RCL2000), and OCL to validate SoD constraints. In that approach, it is checked whether a current state is violated by the authorization constraints. A snapshot based on object diagram is created in order to determine violated constraints. Ray et al. [11] also discuss SoD constraints by using object diagrams and try to alleviate the complexity of OCL. The RBAC constraints that are checked are SoD, prerequisite constraints and cardinality constraints that help imposing a maximum number of role assignments to a user.

2.2 RBAC Constraints and Alloy

Alloy Analyzer [12] emerged as a powerful and rigorous semantic based tool that can be used for precise specification and modeling of a system. Using first order logic, it offers a static structure of the models. Based on specified entities, the set of instances generated by the Alloy analyzer are then checked against the established constraints. The constructs of Alloy are similar to OCL and compatible with it.

To our interest, Alloy should be used for behavioral or dynamic modeling in terms of operations execution. Simulating a system using Alloy involves individual transitions or properties of sequences of transitions. However, use of Alloy for dynamic modeling of security policies has been a scant subject so far. Most of the proposed approaches merely focus on the static analysis where Alloy is used for generating counterexamples against specifications. As an added advantage of Alloy over other languages (e.g., OCL and UML), it is reported [13] as more amenable to automatic analysis. Alloy offers two kinds of automated analysis i.e., *simulation* and *checking*. In simulation, operations are interpreted to compute resulting states, and check that they conform to invariant properties. In checking, Alloy attempts to generate instances of a data structure up to a given (small) maximum size, and can identify counterexamples which don't satisfy a given property. The types of answers that Alloy provides are: *"this property always holds for problems up to size X"* or *"this property does not always hold, and here is a counter example'*.

Regarding the analysis of security models, especially RBAC with constraints, significant amount of work has been carried out using Alloy, mainly based on SoD constraints. Zao [14] has proposed a technique to verify algebraic characteristics of RBAC schema using Alloy. Alloy is used as a constraint analyzer to check inconsistencies among policies. The authors focus on static properties of the security model and don't take into account evolutions of its state. Schaad et al., [15] and Ahn et al. [10] have discussed SoD constraints from RBAC. [15] advocates the suitability of Alloy, and deeply discusses decentralized administration of RBAC and arbitrary changes to a initially stated model that may result in conflicting policies over time *w.r.t*, SoD constraints. They argue that SoD constraints may introduce implicit security policies flaws because of role hierarchies. Several counterexamples are generated to examine the policy which they are interested in. Yu et al. [16] propose scenarios in terms of state transitions to uncover violations in security policies. Such approaches and Alloy analyzer (unlike model checkers) have advantages over model checking tools since model checking merely considers closed-world view of systems [16]; that means no inputs are taken from external resources. In their approach, all operation calls take the form of scenarios and a system state is a configuration of objects. Based on a generated tree (limited depth, limited number of objects, and small domain) of various invocations, scenarios are generated. Using this technique, one can analyze role activation constraints and SoD constraints.

In [17], functional and security models are merged into a single UML model which is translated into Alloy. Alloy can then be used to find a state which

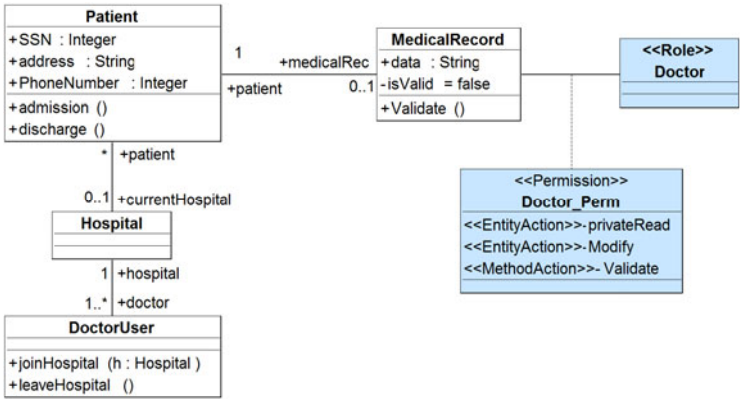

Fig. 1. Functional model enriched by security information (grey shaded classes)

breaks a given property. The properties described in [17] are mainly of static nature, i.e. they focus on the search for a state which breaks a property, and don't search for sequences of actions leading to such a state. Nevertheless, Alloy can take into account the behaviour of the actions of the model, and we believe it has the potential to perform such dynamic analyses.

2.3 SecureMova

The tools presented so far, based on OCL and Alloy, only address the validation of security models, e.g. addressing SoD properties. Most of them don't consider constraints which involve elements of the functional model, and hence don't consider evolutions of the state of the functional model.

In [18], Basin et al report on SecureMova, a tool which supports SecureUML+ComponentUML. The tool allows to create a functional diagram, i.e. a class diagram, and to relate it to permission rules. Constraints can be attached to permissions and may refer to the elements of the functional diagram.

For example, this allows to express the property that doctors may modify a medical record if and only if they are employed by the hospital of the patient. A class *Hospital* may be defined in the functional model (Fig. 1) and relations drawn between *Hospital* and *Patient*, and between *Hospital* and *DoctorUser*. In Sect. 4.2, we will associate an OCL constraint to the permission to access a medical record. This constraint navigates through the functional model to retrieve the *Patient* associated to the medical record, and his/her current *Hospital*. It also retrieves the *DoctorUser* corresponding to the user asking to access the medical record and retrieves his/her associated *Hospital*. Finally, the constraint compares these two hospitals.

With SecureMOVA it is possible to ask questions about a current state, i.e. a given object diagram. Such queries return the actions authorized for a given role, or a given user. They also allow to investigate on overlapping permissions, i.e. permissions which have a common set of associated actions. The tool provides

an extensive set of queries over a given model, possibly associated with a given initial state. All reported examples [18] are of static nature, i.e. they don't allow to sequence actions (either administrative or functional) and check that a given sequence is permitted by the combination of the security and functional models. In the next sections, we will see that a thorough validation of a security policy which includes contextual constraints must also take into account evolutions of the state of the functional model.

3 Motivating Example

Our motivating example is based on the constraint stated above: "If a doctor wants to modify the medical record of a given patient, he must belong to the same hospital as the patient". Let us now consider a malicious doctor, who wants to modify the information of a patient in another hospital. Since the patient and the doctor belong to different hospitals, the doctor will not be permitted to access this information. In order to validate the rules of the security policy, one may try several typical situations and query about the permitted/forbidden actions. Using a tool such as SecureMova, one would provide an object diagram od_1 with one doctor and one patient linked to two different hospitals, and query if the doctor may perform action $setData$ on the patient's medical record. The tool would answer that the doctor is not authorized to perform this action.

Further validation of this security policy should explore dynamic aspects of the policy. For example, is it possible for this malicious doctor to eventually modify the patient's information? Using only static tools, one can check that, given an object diagram od_2 where the malicious doctor belongs to the same hospital as the patient, he will be granted this access. The next question to investigate is: does there exist a sequence of actions which leads a malicious doctor to belong to the same hospital as the patient? This requires to animate a sequence of actions which leads from od_1 to od_2. Such a sequence will presumably call an intermediate operation $joinHospital$ which will link the malicious doctor to the hospital of the patient. Here the dynamic analysis will allow to identify these intermediate actions and check which role has permission to perform these actions.

Another way to group the malicious doctor and the patient in the same hospital is to transfer the patient in the hospital of the doctor. In this second sequence, one should investigate who has the permission to perform such a transfer.

This simple example shows that the validation of a security policy may require dynamic analyses to identify sequences of actions leading to an unwanted state. Moreover, these sequences of actions are not restricted to the standard RBAC functions and may refer to operations defined in the functional model. This is actually the case when constraints referring to the functional model are expressed on permissions. Current tools, such as the ones presented in Sect. 2, which focus on static queries or on the dynamic execution of the sole RBAC functions are not sufficient to perform such dynamic investigations.

4 Using Testing and Verification Techniques

The example of Sect. 3 shows that there is a need for dynamic analyses involving both functional and security models when designing a security policy. Moreover, when the security policy refers to the functional model through the use of constraints, the dynamic analysis should not only cover the RBAC standard functions, but also take into account the behaviour of the functional model.

Dynamic analyses can take two forms: tests and verifications. Tests correspond to the execution of a sequence of actions on the security and functional models, or on their implementations. The test sequence is either defined by the security policy designer, possibly on the basis of use cases, or it may be the output of a test generation tool on the basis of some coverage of the models. Test can contribute to both validation and verification activities. Tests based on use cases correspond to the validation activity because they contribute to show that the security policy meets the users/customers needs. Tests based on model coverage contribute to verification. They can check that the covered behaviours of the model will respect some global properties of the security policy like SoD. Tests also contribute to detect unsatisfiable constraints because such constraints may forbid any state different from the empty state.

Tests can only check a limited number of behaviours. When absolute guarantees are needed, such as ensuring that all threats are handled, verification techniques should be considered. Verification techniques include model-checking and symbolic proof techniques. Both techniques are of interest in the verification of a security policy. Proof techniques can show the existence of some state, and hence prove that constraints are satisfiable, or establish that some property, like SoD, is an invariant of the model. Model-checking is based on model exploration, and can be used to find a sequence of actions leading to a given state or property. In our motivating example, model-checking tools should be experimented to find a path between od_1 and od_2.

4.1 Some Solutions to Explore

Testing techniques require the availability of executable models or implementations. Security policies based on RBAC can easily be made executable, as demonstrated by Sohr in his authorisation editor [9]. Executability of a functional model can be achieved in two ways: either by providing an implementation of the model which can interface with the contextual constraints of the security model, or by providing an executable model. Providing an implementation makes sense in a context where the functional system is designed first, without considering security aspects, and where a security policy must be designed later for this application. It also makes sense during a maintenance phase where a given implemented security policy must evolve. Some prototypes of RBAC can be coupled with an existing implementation. For example, the MotOrBAC tool provides an API between its security engine and the application [19].

The other way is to get an executable functional model. In the case of USE or SecureMova, the model is expressed as a class diagram combined with OCL

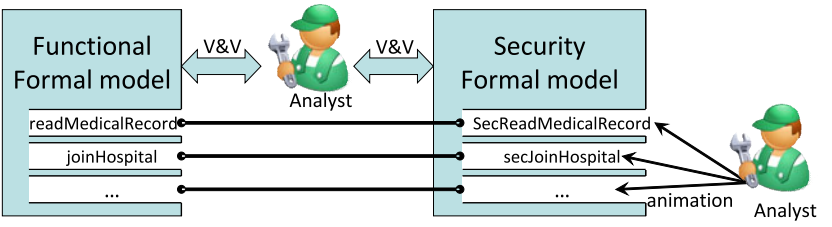

Fig. 2. Analysis of the functional and security models

predicates. In order to turn UML methods into executable ones, one needs to provide an implementation of the methods. Actually, USE allows to define a body for each method using an imperative language based on OCL. It seems that this feature was not explored in [9] and might be interesting to investigate. Another way is to animate the methods based on their pre- and post-conditions. We don't know of tools which support this approach for OCL, but they exist for formal languages such as B[6].

4.2 Using the B Formal Method

The B language actually appears as an interesting option. Several tools have been defined to translate UML models into B specifications; they show at least the feasibility of such translations [20,21]. Regarding the security model, Sohr[9] has already shown that it can be specified in UML+OCL. Since the B language is based on the same principles as OCL (first order predicate logic and set theory), it is possible to propose a similar translation of the security model into B specifications. The B specifications produced from both security and functional models (Fig. 2) can then be analysed using either animation tools such as ProB [22] or proof tools such as Atelier-B[1]. ProB also includes model-checking facilities which can be of interest to search for malicious sequences of operations.

Fig. 2 illustrates the proposed translation of a functional model enriched by an access control policy such as the one of Fig. 1. The functional B specification in the left hand side of Fig. 2 is widely inspired by existing UML to B translation approaches [20,21]. We developed a model driven platform [23] in order to be able to combine and adapt rules proposed by these approaches. The right hand side of Fig. 2 represents the formal specification produced from the security model and which is intended to control the use of the functional B operations. Playing scenarios is done by animating the secured operations (*e.g.* `secReadMedicalRecord`, `secJoinHospital`) which give access uniquely to the authorized functional operations. This approach allows to validate the functional model as well as the security policy. In fact, animation of an authorized operation evolves the state of the functional model and hence allows the analyst to validate both models.

[1] `http://www.atelierb.eu`

Terms "*shallow embedding*" and "*deep embedding*" [24] are often used to describe a mapping between formalisms. The first notion means a direct translation from a source model into a target model, while the second notion means that the mapping leads to structures that represent data types. In the proposed approach we adopt a *shallow embedding* approach when translating the functional model, and a *deep embedding* approach for the security model. In the following we give an overview of the translation principles.

Translation of the Functional Model. As proposed by the existing approaches which transform a UML model into a B specification [20,21] concepts of a UML class diagram lead to sets, variables and relations in the B specification. For example, class *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*.

MACHINE
Functional_Model
SETS
MEDICALRECORD, PATIENT, ...
VARIABLES
MedicalRecord, Patient, isValid, ...
INVARIANT
MedicalRecord $\subseteq$ MEDICALRECORD $\wedge$
isValid $\in$ MedicalRecord $\rightarrow$ **bool**
...
INITIALISATION
MedicalRecord := $\emptyset$
...

Basic operations such as constructors, destructors, getters, setters..., are automatically produced in order to allow state evolution of the functional formal model. We developed a tool which generates all these basic operations and takes into account some basic structural invariants related to mandatory and/or unique attributes, inheritance, composition, multiplicities... The resulting B specification can be enriched in order to take into account less obvious functional constraints and also to add manually other functional operations such as operation *Validate* of Fig. 1. Proof and animation tools can then be used in order to analyse the correctness of the functional model independently from security aspects. Let us consider for example a functional OCL constraint which indicates that a patient can't leave an hospital if his medical record is not validated:

```
context MedicalRecord inv MR_Validation:
    self.isValid = FALSE
    implies self.Patient.currentHospital -> notEmpty()
```

In other words, this constraint considers 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 translated in B as follows:

$$\forall p \cdot (p \in Patient \ \wedge \ isValid(PatientMedicalRecordRel^{-1}(p)) = FALSE) \\ \Rightarrow PatientHospitalRel[\{p\}] \neq \emptyset$$

Relations *PatientMedicalRecordRel* and *PatientHospitalRel* are issued respectively from the association between class *Patient* and class *MedicalRecord* and the association which links class *Patient* to class *Hospital*. Taking into account this invariant leads to improvements of the functional model. For example, operation *setData*, which is a basic setter, modifies attribute *data* of a *MedicalRecord*, and also turns the attribute *isValid* into false. Hence, precondition of operation *setData* must be enforced in order to avoid a violation of the previous invariant when trying to modify data of a medical record of a patient who is not admitted in any hospital.

$setData(mr, dd) \triangleq$
PRE
$mr \in MedicalRecord \ \wedge \ dd \in TheData$
$\wedge \ PatientMedicalRecordRel(mr) \in \mathbf{dom}(PatientHospitalRel)$
THEN
$isValid(mr) := FALSE \ \ ||$
$data(mr) := dd$
END;

Such validations are done using the AtelierB tool which allows to prove the functional model consistency with its invariants.

Translation of the Security Model. Translation of the security model follows a *deep embedding* approach. Indeed, we propose a B formalization of a variant of the secureUML meta-model [2]. Elements of a security model are then directly injected in this B specification. The access to the operational part of the functional formal model is controlled by the B specification issued from the security model. As shown in Fig. 2, we associate to each functional operation a secured operation in the security formal model which verifies that a user has permission to call a functional operation. For example, operation *secure_setData* is intended to verify accesses to operation *setData* above.

$secure\_setData(mr, data) \triangleq$
PRE
$mr \in MedicalRecord \wedge data \in TheData$
THEN
SELECT
$MedicalRecord\_setData \in isPermitted[currentRole]$
THEN
$setData(mr, data)$
END
END;

Variable *currentRole* contains the set of roles activated by a user in a session. Set *isPermitted* computes for each role the set of authorized functional operations. Then, the guard $MedicalRecord\_setData \in isPermitted[currentRole]$ verifies whether *setData* is allowed to the connected user using his active roles. Fig. 1 grants to doctors the permission to call modify operations such as *setData*. Hence, animation of *secure_setData* shows that every doctor may modify medical records. In Sect. 2.3, we considered an additional constraint saying that in order to modify a medical record, the doctor must be employed by the current hospital of the patient. This constraint can be expressed in OCL as follows:

```
context Doctor_Perm::Modify inv :
    session.user.isTypeOf(DoctorUser) implies
    session.user.Hospital = self.medicalRecord.Patient.currentHospital
```

As this constraint is expressed in the context of a modification permission (*i.e. Doctor_Perm*), then we take it into account in our formal specifications by strengthening the guard of *secure_setData* as follows:

$$
\begin{array}{l}
MedicalRecord\_setData \in isPermitted[currentRole] \ \wedge \\
(currentUser \in DoctorUser \\
\quad \Rightarrow HospitalDoctorRel(currentUser) \\
\qquad = PatientHospitalRel(PatientMedicalRecordRel(mr)))
\end{array}
$$

Now, animation of *secure_setData* will fail if the doctor and the patient are linked to different hospitals. This constraint shows the impact of the functional model on the security model because the guard of the secured operation navigates through the functional model state in order to retrieve and compare the hospital in which the patient is admitted and the hospital of the connected doctor.

4.3 Support of History-Based Constraints

The constraints expressed in OCL only refer to a given instant of time. When expressing history-based constraints, it is necessary to refer to several distinct instants of time. For example, consider the following rule: "If a patient has left the hospital, all doctors belonging to the hospital during the patient's stay will keep read access to his medical record.". If we want to express this rule as a read permission associated to an OCL constraint, we need to extend the functional model with information about past states, and this information is for security's sake only. This goes against separation of concerns.

In [9], Sohr suggests the use of TOCL, an extension of OCL with temporal operators. Although this provides a way to express history based constraints, it appears that no tool is currently available to support the use of this formalism.

In the sequel, we address history-based constraints using process algebra and the B method. Process algebra can be used to model workflows of actions and all ordering and security constraints related to a dynamic security policy.

4.4 The ASTD Notation

In order to specify information systems, the EB^3 [25] method was developed. It features a process algebra similar to CSP [26] with some IS-oriented additions such as quantifications. However, it lacks a graphical representation that can help during the modeling process and that is one of the advantages of UML statecharts. The ASTD notation [27] is a graphical representation linked to a formal semantics created to specify systems such as IS. An ASTD (e.g. Fig. 3) defines a set of traces of actions accepted by the system. ASTD was introduced as an extension of Harel's Statecharts [28] and is based on operators from EB^3. An ASTD is built from transitions, denoting action labels (i.e. method calls) and parameters, and places that can be elementary (as states in automata theory) or

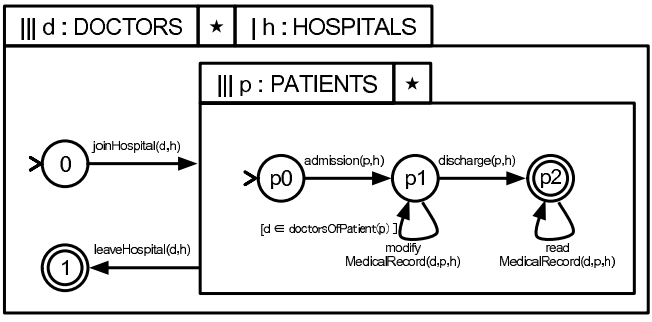

Fig. 3. An ASTD rule expressing ordering constraints in an hospital IS

ASTD themselves. Each ASTD has a type associated to a formal semantics. One of the main features of ASTD is to allow parameterized instances and quantifications, aspects missing in original statecharts. This means that ASTD can describe not only the behavior of one instance but also the behavior of sets of entities and relationships of a system. A formal description of the ASTD notation is given in [29].

In the example of the hospital IS, ASTD can help modeling properties such as "If a patient has left the hospital, all doctors belonging to the hospital during the patient's stay will keep read access to his medical record". The ASTD of Fig. 3 expresses that for all doctors (||| d : DOCTORS), at a given time there is a unique hospital where the doctor is affected (| h : HOSPITALS). This hospital can change during the doctor career (thanks to the * operator). Once the doctor d joined an hospital h, he can modify medical records of patient p if p was admitted in hospital h and if he is one of his/her doctors. If p leaves the hospital, d can still read his/her medical records, unless d leaves the hospital. At any time, d can leave the hospital, loosing its reading rights over medical records.

4.5 Using ASTD to Validate a Policy

ASTD models are executable using the *i*ASTD [30], an interpreter for ASTD. *i*ASTD efficiently determines if actions can be executed by the model and computes the ASTD state after the execution. *i*ASTD accepts as inputs 2 files. The first one is a description of the topology of the ASTD model to validate and the second one is a trace of actions with their parameters to be executed. After parsing both files, *i*ASTD computes the initial state of the ASTD and then tries to execute the first action of the trace. If the execution succeeds, *i*ASTD computes the new state of the ASTD and goes on to the next action of the trace to be executed. If the execution is not allowed, *i*ASTD does not modify the current state of the ASTD and warns the user of refusal of the action. *i*ASTD then tries to execute the next action in the trace. Validation of an ASTD is thus performed by providing scenarios (traces) that should be accepted and others that should be rejected. Such scenarios can be generated in order to check usual access-control policy properties or requirements.

In the example presented in Fig. 3, we would like to grant doctors linked a given hospital the right to read medical records of patients only if they were linked to this hospital during the patient stay. We can validate our policy by executing some scenarios over the model. Executing the trace { joinHospital($Alex, H1$) ; admission($Bob, H1$) ; discharge($Bob, H1$) ; readMR($Alex, Bob, H1$) } will check that a doctor linked to hospital $H1$ can read the record of a patient linked to the same hospital in the same time interval. With another scenario such as { joinHospital($Alex, H1$) ; admission($Bob, H1$) ; discharge($Bob, H1$) ; leaveHospital($Alex, H1$) ; readMR($Alex, Bob, H1$) }, the interpreter will reject the last action because $Alex$ is not linked to the hospital at the time he attempts to read the medical record. However the scenario { admission($Bob, H1$) ; joinHospital($Alex, H1$) ; discharge($Bob, H1$) ; readMR($Alex, Bob, H1$) } should be accepted by our policy, but will be rejected by iASTD since $Alex$ joined the hospital after the admission of Bob. Our security policy is hence too strict and should be adapted in order to accept this scenario. This example shows how to use iASTD and use case scenarios in order to validate ASTD specifications which define history-based constraints on security policies.

4.6 Putting It All Together

In Sect. 4.2, we propose a set of rules to produce B specifications from both the functional part of the system and the static access control model. Translation rules from ASTD models into event-B have also been defined in [31]. Hence, the proposed methodology for modeling IS security starts from a set of graphical models: (*i*) a class diagram for the functional model, (*ii*) a static security policy linked to the class diagram and (*iii*) ASTD models describing a set of allowed traces of actions. These graphical models are used for specifying, visualizing, understanding and documenting a security policy. In order to rigorously check their correctness, a formal B specification can be derived. Thus, a functional model and its associated access control policy, including both static and dynamic aspects, are specified using the same formal notation. We extensively use the inclusion mechanism of B to link secured operations with functional operations. This allows the consistency of the whole system to be formally checked, activity that can be assisted by the tools associated to the B method.

5 Conclusion

This paper has addressed the validation of security policies which include contextual constraints refering to the functional model of an information system. These are essential activities when designing or modifying a security policy. Separation of concerns suggests to treat the functional and security models in isolation. Unfortunately, when constraints establish a link between these models, validation activities must consider both security and functional models, because changes in the state of the latter may grant or deny permissions in the former. In Sect. 2, we have stated that most tools focus on the validation of the security model and can't take into account constraints which link it to the functional model. Only

a few tools, like SecureMova, take both models into account, but their analyses are of static nature, and don't support evolutions of the state of the functional model. In Sect. 3, a motivating example has illustrated the need for dynamic analyses which take into account both models. Our example includes properties whose context is either the current functional state, or the history of this state.

We have proposed an approach, based on the B method and ASTD, where such contextual constraints may be expressed, and included in a formal model. This model can be animated with user defined scenarios and the constraints are evaluated during this animation. The scenarios are defined during requirements analysis, and can be shown to the customer, contributing to the validation of both functional and security models. Playing these scenarios may reveal that the security policy is too strict and forbids normal behaviours. Scenarios may also correspond to potential attacks, and help evaluate the capabilities of the security policy to detect and prevent such attacks.

Our current work implements the tools associated to this approach and studies the interactions between state-based and history-based constraints. This toolset will then be evaluated on case studies of the ANR Selkis Project[2].

References

1. Ferraiolo, D.F., Kuhn, D.R., Chandramouli, R.: Role-Based Access Control. Computer Security Series. Artech House, Boston (2003)
2. Basin, D.A., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Transaction of Software Engineering Methodology 15(1), 39–91 (2006)
3. Clark, D.D., Wilson, D.R.: A comparison of commercial and military computer security policies. In: IEEE Symposium on Security and Privacy, pp. 184–195 (1987)
4. Fernández, E.B.: A methodology for secure software design. In: Proc. of the Int. Conf. on Software Engineering Research and Practice, SERP 2004, pp. 130–136. CSREA Press (2004)
5. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2004)
6. Abrial, J.: The B-Book. Cambridge University Press, Cambridge (1996)
7. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Precise Modeling With UML. Addison-Wesley, London (1998)
8. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3), 27–34 (2007)
9. Sohr, K., Drouineaud, M., Ahn, G.J., Gogolla, M.: Analyzing and managing role-based access control policies. IEEE Trans. Knowl. Data Eng. 20(7), 924–939 (2008)
10. Ahn, G., Hu, H.: Towards realizing a formal RBAC model in real systems. In: 12th ACM Symp. on Access Control Models and Technologies. ACM Press, New York (2007)
11. Ray, I., Li, N., France, R.: Using UML to visualize role-based access-control constraints. In: Proceedings of the 9th ACM Symposium on Access Control Models and Technologies, pp. 115–124. ACM Press, New York (2004)
12. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256–290 (2002)

[2] `http://lacl.univ-paris12.fr/selkis/`

13. Power, D., Slaymaker, M., Simpson, A.: On the modelling and analysis of amazon web services access policies. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 394–394. Springer, Heidelberg (2010)
14. Zao, J., Wee, H., Chu, J., Jackson, D.: RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis. In: Proceedings of 8th ACM Symposium on Access Control Models and Technologies (2003)
15. Schaad, A., Moffett, J.D.: A lightweight approach to specification and analysis of role-based access control extensions. In: Proc. of 7th SACMAT. ACM Press, New York (2002)
16. Yu, L., France, R., Ray, I., Ghosh, S.: A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In: Int. Conf. on Engineering Complex Computer Systems. IEEE, Los Alamitos (2009)
17. Toahchoodee, M., Ray, I., Anastasakis, K., Georg, G., Bordbar, B.: Ensuring spatio-temporal access control for real-world applications. In: 14th ACM Symp. on Access Control Models and Technologies, SACMAT 2009. ACM, New York (2009)
18. Basin, D.A., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Information & Software Technology 51(5), 815–831 (2009)
19. Autrel, F., Cuppens, F., Cuppens-Boulahia, N., Coma-Brebel, C.: MotOrBAC 2: a security policy tool. In: SARSSI 2008: 3e Conf. sur la Sécurité des Architectures Réseaux et des Systèmes d'Information, (Télécom Bretagne) (2008)
20. Mammar, A., Laleau, R.: From a B formal specification to an executable code: application to the relational database domain. Inf. Softw. Technol. 48, 253–279 (2006)
21. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Transactions on Software Engineering Methodology 15(1), 92–122 (2006)
22. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT 10(2), 185–203 (2008)
23. Idani, A., Labiadh, M.A., Ledru, Y.: 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)
24. Wildmoser, M., Nipkow, T.: Certifying Machine Code Safety: Shallow versus Deep Embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 305–320. Springer, Heidelberg (2004)
25. Frappier, M., St-Denis, R.: EB^3: an entity-based black-box specification method for information systems. Software and Systems Modeling 2(2), 134–149 (2003)
26. Hoare, C.A.R.: CSP–Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
27. Frappier, M., Gervais, F., Laleau, R., Fraikin, B., St-Denis, R.: Extending statecharts with process algebra operators. Innovations in Systems and Software Engineering 4(3), 285–292 (2008)
28. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
29. Frappier, M., Gervais, F., Laleau, R., Fraikin, B.: Algebraic state transition diagrams. Technical Report 24, Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada (June 2008)
30. Salabert, K., Milhau, J., et al.: iASTD: un interpréteur pour les ASTD. In: AFADL 2010, Poitiers, France (2010)
31. Milhau, J., Frappier, M., Gervais, F., Laleau, R.: Systematic translation rules from ASTD to event-B. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 245–259. Springer, Heidelberg (2010)