Automated Reviewing of Healthcare Security Policies

Nafees Qamar[1], Johannes Faber[1], Yves Ledru[2], and Zhiming Liu[1]

[1] United Nations University
International Institute for Software Technology
{nqamar,jfaber,lzm}@iist.unu.edu
[2] UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble2/CNRS, LIG
yves.ledru@imag.fr

Abstract. We present a new formal validation method for healthcare security policies in the form of feedback-based queries to ensure an answer to the question of *Who* is accessing *What* in Electronic Health Records. To this end, we consider Role-based Access Control (RBAC) that offers the flexibility to specify the users, roles, permissions, actions, and the objects to secure. We use the Z notation both for formal specification of RBAC security policies and for queries aimed at reviewing these security policies. To ease the effort in creating the correct specification of the security policies, RBAC-based graphical models (such as SecureUML) are used and automatically translated into the corresponding Z specifications. These specifications are then animated using the Jaza tool to execute queries against the specification of security policies. Through this process, it is automatically detected who will gain access to the medical record of the patient and which information will be exposed to that system user.

1 Introduction

Security and privacy in information systems generally concern questions such as *Who accesses What*. An Electronic Health Record (EHR) is a longitudinal electronic record of health information for a patient generated by one or more encounters in any care delivery setting. Security mechanisms are supposed to be applied to protect such EHRs to shield against *external threats* from outside the system, such as attacking or running malicious applications, as well as *internal threats* from inside the system, e.g., a valid system user illegitimately accesses private data of a patient. It was, however, reported that major threats to patient privacy actually stemmed mostly from internal factors [Jou09]. For this reason, it is essential to investigate who is doing what in a system besides ensuring smooth data availability. For example, what are the operations a user such as a nurse can perform, and what resources are accessible by a user? The system also needs to be flexible enough to allow exceptional access to system resources, especially in medical emergency cases. Due to these exceptions, the large number of stakeholders, end users, and interaction components, the specification of policies

J. Weber and I. Perseil (Eds.): FHIES 2012, LNCS 7789, pp. 176–193, 2013.
© Springer-Verlag Berlin Heidelberg 2013

on security and privacy and their correct implementation in EHR systems are particularly challenging. Not surprisingly, medical data disclosure is the second highest reported breach [HY06]. To address this issue, legal regulations, such as USA's Health Care Insurance Portability and Accountability Act (HIPAA), are in place as safeguards to confidentiality, integrity, and availability of these systems.

To guarantee that application-specific security objectives are enforced by the enacted security policies based on regulations like HIPAA, the rules that define the security policies need to be justified using validation techniques. To this end, we propose using the Z notation [Spi92, DW96] to represent the security rules as well as to specify queries for revealing internal threats to EHRs. The execution of the queries animates the security policies to generate feedbacks on authorized and unauthorized states of an EHR system. Additionally, these also help in analyzing over- or under-designed security policies, which otherwise can block desirable operations or permit undesirable ones, respectively.

To ease the effort in model construction and understanding, we propose to use a graphical notation, such as SecureUML [LBD02] or UMLsec [Jür05], to model security policies. The graphical models are automatically translated into specifications in the formal Z notation, which is amenable to formal analysis. In this way, the proposed formal approach can be integrated with graphical modeling and transformation techniques – the so-called model-driven techniques. For specifying security policies, we apply the standardized Role-based Access Control (RBAC) mechanism [FSG+01]. RBAC offers roles, which are permanent organizational positions whilst users can arbitrarily be changed. Based on this, we introduce formalized review functions in terms of Z specifications. These review functions can be animated using, e.g., the Jaza tool to get feedback on which actions are available to which role, on access to specific resources, and on duplicate permissions. To further reflect the complex situation in the healthcare domain, we additionally introduce Separation of Duties (SoD) constraints to analyze conflicts of interest between different stakeholders in the system. This is done statically, by ensuring that a user cannot have two conflicting roles at all, as well as dynamically, by ensuring that a user cannot have two conflicting roles in a single session. We exemplary demonstrate that this approach is suitable to deal with emergency situations, where immediate access to information might be needed and report on the types of information one needs to collect before deploying a security policy. We prefer RBAC to Mandatory Access Control [BL75] and Discretionary Access Control [DOD85], because it builds on a generic principle of access control that makes it adaptable to any organizational structure and flexible with respect to the application implementation.

To summarize, we present a new formal validation method for healthcare security policies with two main contributions. First, we introduce formally specified queries for automatically reviewing security policies and analyze what information of an EHR is exposed to a system user. Second, we introduce SoD constraints to cope with the complex security policies usually found in EHR systems. We use an example from the healthcare domain for demonstration.

The paper is structured as follows: Section 2 discusses state-of-the-art formal techniques on security policies. Section 3 states RBAC and gives a scenario, which is used as a running example throughout the paper. Section 4 demonstrates the formalization of security policies in Z notation, whereas Sect. 5 presents formally specified queries for reviewing security policies. Section 6 formalizes SoD constraints, and finally, Sect. 7 concludes and shares some perspectives.

2 Related Work

Healthcare privacy and security techniques are intended to cope with a number of issues such as authorized data disclosure, integrity of information, regulatory implication for healthcare, and information security risk management. The survey [AJ10] also considers a multitude of such techniques and provides a classification. The authors conclude that existing techniques are inadequate to meet security and privacy challenges in EHRs. It also shows that healthcare security and privacy issues have not been treated in a deserved manner. Our findings complement this survey concerning these issues, which also have not been dealt with formal languages such as the Z notation, generally known for their precision and unambiguity. For example, [TMB06] uses formal methods to improve medical protocols, but such techniques are missing in general.

The standardized Z notation [Spi92] has been successfully applied to various industrial projects for formal modeling and development [Bow03]. Our past work presents an RBAC-based security kernel using Z [QLI11, LQI+11], which shows that Z can be effectively used to verify or validate security policies. The work in this paper builds on these preliminary results with an extended set of formal queries for validation of security policies and formal SoD constraints. The use of Z in our previous work is motivated by the support of authorization constraints, which do not appear in this paper. When there are authorization constraints, one needs to consider the evolution of the state of the functional model [LIM+11]. Currently, only our tool takes this evolution in the animation of the complete model into account.

Alloy has the potential to do similar work as Z, but currently no tool is available covering both functional security models and authorization constraints. Regarding B, a similar tool is currently under development in the Selkis project [MIL+11]. The tool will allow using the ProB tool both for animation and model checking.

ISO-standardized RBAC has widely been described by researchers using Z such as [Hal94, AK06, YHHZ06]. However, the work there offers only generic formal representation of RBAC. There are other techniques for validation and verification of security properties [MSGC07, Bos95, AK06]. In particular, [MSGC07] proposes a process to verify Z specifications by the Z/EVES theorem prover. In parallel, OCL expressions are also meant to specify restrictions on a system model but do not support feedback queries. For example, SecureMOVA [BCDE09] offers a set of queries to analyze security policies expressed as formulas in UML's Object Constraint Language.

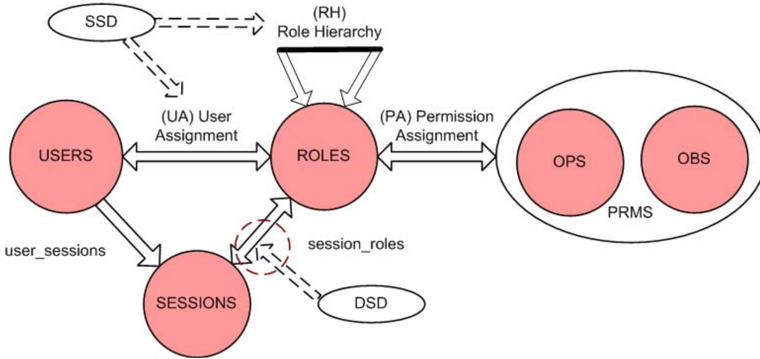

Fig. 1. Role Based Access Control [FSG+01]

The work [ZWCJ02] proposes to verify algebraic characteristics of RBAC schemas using Alloy. Alloy is used as a constraint analyzer to check the inconsistencies among roles and Static Separation of Duty (SSD) constraints and to generate a counter-example when inconsistencies are found. However, the work addresses SSD constraints only. [SM02, AH07] also discuss SoD constraints. The former discusses decentralized administration of RBAC and allows arbitrary changes to an initially stated model that may result in conflicting policies over time with respect to SoD constraints. They argue that SoD constraints may introduce implicit security policy flaws because of role hierarchies. In [TRA+09], a translation from UML to Alloy is given for verification of the UML models. The work is mainly focused on analysis of contextual information such as location and time for access decisions.

3 Role-Based Access Control

In this section, we introduce the preliminaries of Role-based Access Control (RBAC). We will use a running example to illustrate the basic concepts and properties.

3.1 Data Model of RBAC

The data model of RBAC [FSG+01] as shown in Fig. 1 is based on five data types: users (USERS), roles (ROLES), objects (OBS), permissions (PRMS) and executable operations (OPS) by users on objects. A sixth data type for sessions (SESSIONS) is used to associate roles temporarily to users. Sessions correspond to the dynamic aspect of RBAC that actually includes session management.

RBAC differentiates between users and roles. A role is considered as a permanent position in an organization whereas a given user might be switched with another user for that role. Thus, *rights* are offered to roles instead of users. Roles are assigned to permissions that can later be exercised by users playing

these roles. Modeled objects (OBS) in RBAC are potential resources to protect. Operations (OPS) are viewed as application-specific user functions. Other constructs included in the model are user assignment, hierarchy, and permission assignment, which are designated as UA, RH, and PA, respectively. Here, we very briefly outline all the aforementioned RBAC constructs:

- *User* is a person who uses a system or an automated agent.
- *Role* is an organization entity or a permanent position in an enterprise. Each role may have an allowable set of actions according to the access control policy. In this way, access to computational resources is realized via roles.
- $UA \subseteq USERS \times ROLES$ is a many-to-many mapping between users and roles; UA specifies which roles can be taken by a given user.
- $PA \subseteq PRMS \times ROLES$ is a many-to-many mapping permission-to-role; PA expresses which roles may be granted a given permission.
- $user_sessions(u : USERS) \rightarrow 2^{SESSIONS}$ is a mapping of user u onto a set of sessions; it lists the current sessions of a given user.
- $session_roles(s : SESSIONS) \rightarrow 2^{ROLES}$ is a mapping of session s onto a set of roles; it lists the current roles of a given user in a given session.
- $RH \subseteq ROLES \times ROLES$ is a partially ordered role hierarchy; a senior role may inherit the permissions from its junior roles.
- $PRMS : 2^{OPS \times OBS}$ is a set of permissions. Permissions are regarded as an approval to perform operations on RBAC-protected objects. An executable image of a program is considered as an operation, which executes some function for the user. For example, within medical records, operations might include insert, delete, append, and update medical instructions.

3.2 Example: RBAC-Based Security Management for EHRs

Healthcare security policies are ideally modeled using RBAC, because permanent positions such as doctors, nurses, and other healthcare staff can be mapped to the roles as set in the policy. Figure 2 represents a small healthcare security policy management, which attempts to secure medical records by the use of roles associated to permissions given as stereotypes, while permissions are on the class i.e., MedicalRecords. For instance, a patient, which corresponds to a particular role in an EHR system, has the ability to read his/her own medical record. A doctor inherits permissions from the patient, besides holding another permission given as UserCredentials, on which the doctor could exercise a write operation. The role EmergencyOfficer is assigned with a permission such as read, which is intended for emergency access. Using our toolset [LQI+11] one can translate such diagram into a Z model. Here we confine ourselves to explain the needed part to allow reviewing of such formal translations.

Access Control Violations. Access control rules specified as graphical models are hard to validate because of their ambiguous semantics. This in turn leads to information integrity and confidentiality problems in general. In the example

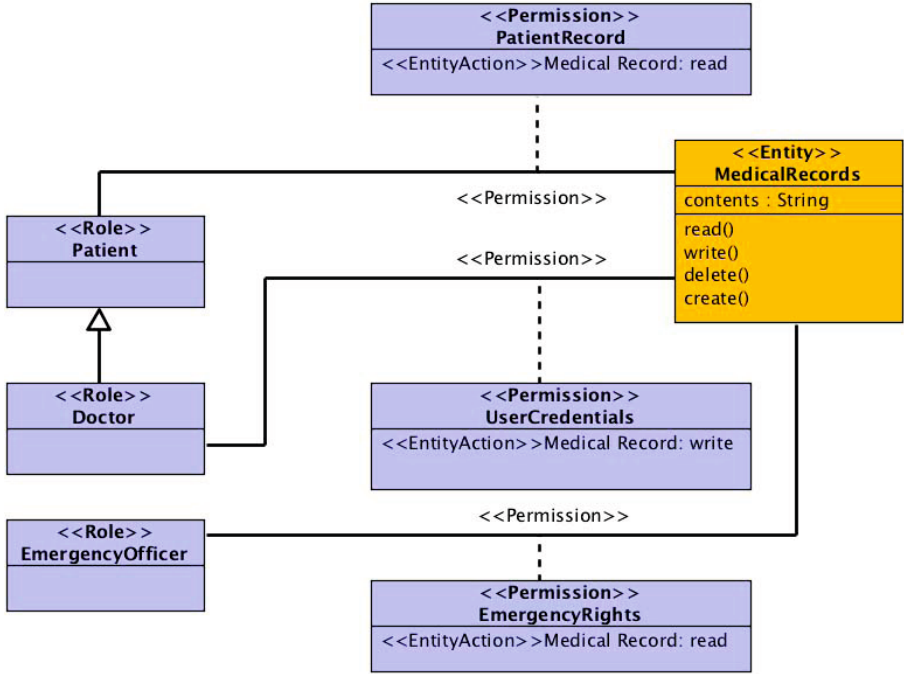

Fig. 2. Design of a medical application using SecureUML

above, a patient could change his/her own medical record if the security policy is not correctly realized in the system. Similarly, a doctor being also a patient can forge his/her own medical record if not avoided by a corresponding SoD constraint. To address these inadequate security policies, we introduce a technique for automatically reviewing such deficiencies in Sect. 5 after the next section's discussion on the formalization of policies.

4 Formalized Healthcare Security Policies

This section gives an overview of the formalization of security policies, which has been introduced in our previous work [QLI11].

4.1 The Z Notation and the Jaza Tool

The ISO-standardized Z language [Spi92] offers an extensive set of concepts and constructs from first-order logic and set theory to specify software systems. Schemas are the major structuring primitives in Z. Each schema is further divided into two components: the signature part, which includes variables and types, and the predicate part, for imposing constraints upon these variables.

$[PERMISSION, SESSION, USER]$
$ROLE ::= Patient \mid Doctor \mid EmergencyOfficer$
$RESOURCE ::= MedicalRecords$
$ATOMIC_ACTION ::= Read \mid Write \mid Delete \mid Create$

Sets
$role : \mathbb{F}\ ROLE$
$user : \mathbb{F}\ USER$
$session : \mathbb{F}\ SESSION$
$resource : \mathbb{F}\ RESOURCE$
$permission : \mathbb{F}\ PERMISSION$
$atm_action : \mathbb{F}\ ATOMIC_ACTION$

Fig. 3. Z types for security policies

A schema represents an operation, and it may reference further schemas by means of their names. We will use the running example to explain the Z constructs used here.

Jaza (`http://www.cs.waikato.ac.nz/~marku/jaza/`) can animate a large subset of constructs of the Z language. It uses a combination of rewriting and constraint solving to find final states for a given initial state. If the initial state does not satisfy the precondition of the operation, the tool returns "No Solutions". The tool can be further queried to find out which constraint could not be satisfied.

4.2 Z Models for Security Policies

We explain our formalization of security policies with the Z notation following the example from Sect. 3.2. This approach from [QLI11] can be used to formalize security policies following the RBAC data model as given in Sect. 3.1.

Z Schemas. Using Z, six types are introduced in Fig. 3 as basic type definitions (PERMISSION, SESSION, USER) or enumerated types on the left. The value of these types is based on the security model presented in Fig. 2. The schema Sets on the right side declares corresponding finite sets ($\mathbb{F}$) for each of these types.

Jaza Representation. The following Jaza expressions initialize some of these sets according to the values of the running example from Sect. 3.2.

```
atm_action' == {Read, Write, Delete, Create},
permission' == {"PatientRecord", "UserCredentials", "EmergencyRights"},
resource' == {MedicalRecords},
role' == {Patient, Doctor, EmergencyOfficer},
user' == {"Alice", "Bob", "Mark"},
```

The schema Perm_Assignment reminds of the underlying translation from graphical SecureUML models to Z notation. It is used to compute the table

perm_Assignment in Fig. 4. In [QLI11] this schema as well as the translation of SecureUML models in general are explained. One can also find there the corresponding rules to automatically generate other RBAC structures such as PA, UA, sessions, and role hierarchy. For this work, it is sufficient to understand the type of the resulting permission assignment as shown in the following schema: users with assigned roles are related to a set of permissions for specific resources.

Perm_Assignment

...

$perm_Assignment : (USERID \times USER \times ROLE) \leftrightarrow (PERMISSION \times ATOMIC_ACTION \times RESOURCE)$

...

The table perm_Assignment, pictured in Fig. 4 in Jaza syntax, results from the translation process. It creates a link between a user's ID, users and their assigned roles to the permissions, the operations, and the resources. The set for user IDs (USERID) is not available in RBAC, but we believe it will be useful when implementing a real system. This generated table assigns the initial permission values for the use with a Z-based formal model animator such as Jaza.

This sums up the formalization of the access control information for the running example, which can be interpreted by a tool. In the following, we present formal queries allowing us to analyze suchlike formal models for security rules.

5 Formal Queries for Healthcare Security Policies

The RBAC model provides mainly three types of functions to operate on security policies: administrative, supporting system, and review. *Administrative functions* involve creation and maintenance of basic sets of elements. These sets are USERS, ROLES, OPS and OBS. Additionally, constructing relations among the sets is also supported by administrative functions (UA and PA assignments). This has already been covered in an earlier paper [QLI11].

Review Functions, on the other hand, help in querying the data structures such as those of UA and PA assignments. The administrator may view the contents of specified relations through review functions. By this means, we can perform queries to request the users assigned to a role, permissions of a role, and allowed roles in a session. In the RBAC standard, the review functions are either mandatory, like querying the assigned users and assigned roles, or optional, like querying permissions of a role. Therefore, not all RBAC implementations provide all review functions. In the following, we present a set of formalized and extended review functions, which provide feedback on the contents of the security policies.

5.1 Authorized Roles for an Atomic Action

The first operation schema EvaluateRoleAuthorizedAtomicAction computes the set of atomic actions for a given role. This query is helpful when one needs to evaluate who can perform a particular operation in a given security policy.

$$
\begin{array}{l}
\textit{EvaluateRoleAuthorizedAtomicAction} \\
\hline
\Xi \textit{Sets}\,;\ \Xi \textit{ComputeAssignment} \\
\textit{role}? : \textit{ROLE} \\
\textit{atomicActions}! : \textit{ROLE} \leftrightarrow \\
\qquad\qquad (\textit{PERMISSION} \times \textit{ATOMIC_ACTION} \times \textit{RESOURCE}) \\
\hline
\textit{role}? \in \operatorname{dom} \textit{concrete_Assignment} \\
\textit{atomicActions}! = \{ \textit{prm} : \operatorname{ran} \textit{concrete_Assignment} \mid \\
\qquad (\textit{role}?, \textit{prm}) \in \textit{concrete_Assignment} \bullet (\textit{role}? \mapsto \textit{prm}) \} \\
\hline
\end{array}
$$

The declaration part of a Z schema is notated above the horizontal line, whereas the predicate part below the horizontal line defines constraints on the declared variables. The declaration part of the schema EvaluateRoleAuthorizedAtomicAction includes the state schema Sets (the symbol Ξ basically indicates that its elements are not changed in this schema) and an input variable role? of type ROLE. The output set being computed, atomicActions! (i.e., operations from a modeled system), is a relation that is a cross product of a role with associated permissions, atomic actions, and resources. The predicate part of the schema checks that the input role (role?) is actually from the domain of the relation concrete_Assignment. The output (atomicActions!) is the set of all possible values associated with a particular role (i.e., role?). Note that concrete_Assignment is actually defined in a further schema ComputeAssignment, which is not shown here in full detail due to space reasons, but can be found in [QLI11].

$$
\begin{array}{l}
\textit{ComputeAssignment} \\
\hline
\ldots \\
\textit{concrete_Assignment} : \textit{ROLE} \leftrightarrow \\
\qquad\qquad (\textit{PERMISSION} \times \textit{ATOMIC_ACTION} \times \textit{RESOURCE}) \\
\hline
\ldots \\
\hline
\end{array}
$$

The set construction in schema EvaluateRoleAuthorizedAtomicAction

$$
\{ \textit{prm} : \operatorname{ran} \textit{concrete_Assignment} \mid \\
\qquad (\textit{role}?, \textit{prm}) \in \textit{concrete_Assignment} \bullet (\textit{role}? \mapsto \textit{prm}) \}
$$

first declares a local variable $\textit{prm}$ to be in the range of $\textit{concrete_Assignment}$. The predicate part, $(\textit{role}?, \textit{prm}) \in \textit{concrete_Assignment}$, selects tuples of the shape $(\textit{role}?, \textit{prm})$ occurring in $\textit{concrete_Assignment}$. Finally, the expression behind

```
perm_Assignment ==
{
  (("ABC001", "Alice", Patient, ("PatientRecord", Read, MedicalRecords)),
  (("ABC002", "Bob", Doctor, ("PatientRecord", Read, MedicalRecords)),
  (("ABC003", "Bob", Doctor, ("UserCredentials", Write, MedicalRecords)),
  (("ABC004", " Mark", EmergencyOfficer,
                              ("EmergencyRights", Read, MedicalRecords)),
},
```

Fig. 4. Permission assignment in Jaza syntax

the • symbol collects the maplets for all of these permissions in the set. By this the result relation is built. Below we show an example of executing this schema against an input role EmergencyOfficer from the running example (cf. Fig. 4). Jaza lists the operations that the emergency officer is permitted to perform.

```
JAZA> ;EvaluateRoleAuthorizedAtomicAction
  Input role? = EmergencyOfficer
  atomicActions!==
  {(EmergencyOfficer, ("EmergencyRights", Read, MedicalRecords))}
```

5.2 Actions Available for a Role

The operation schema EvaluateActionsAgainstRoles works exactly opposite to the operation schema EvaluateRoleAuthorizedAtomicAction, which, for a given atomic action, returns the list of all associated roles (along with a resource and a permission) to perform that action.

EvaluateActionsAgainstRoles

$$\Xi Sets;\ \Xi ComputeAssignment$$
$$atm_action? : ATOMIC_ACTION$$
$$roleAction! : ROLE \leftrightarrow (PERMISSION \times ATOMIC_ACTION \times RESOURCE)$$

$$roleAction! = \{r : \mathrm{dom}\ comp_Assignment;\ p : permission;\ rsrc : resource \mid (r \mapsto (p, atm_action?, rsrc)) \in concrete_Assignment \bullet (r \mapsto (p, atm_action?, rsrc))\}$$

The input (atm_action?) is of the set type ATOMIC_ACTION. The output roleAction! has the same type as given in the previous schema. The set roleAction! retrieves the allowed roles to perform an atomic action. Note that we also retrieve the associated permissions and resources with each obtained role since this appears more comprehensible from a security engineer's point of view. In the following example of this schema query, we provide an atomic action named Read, and the corresponding information is returned.

```
JAZA> ;EvaluateActionsAgainstRoles
  Input atm\_action? = Read
  atomicActions!=={(Patient, ("PatientRecord", Read, MedicalRecords)),
  (Doctor, ("PatientRecord", Read, MedicalRecords)), (EmergencyOfficer,
  ("EmergencyRights", Read, MedicalRecords))}
```

5.3 Analyzing Access to a Resource

It is equally important to know the resources within the system that can be accessed by some roles. EvaluateResourcesAccess is used to this end. For a given resource resource?, it returns the pairs of atomic actions associated with that particular resource.

$$
\begin{array}{l}
\textit{EvaluateResourcesAccess} \\
\hline
\Xi \mathit{Sets};\ \Xi \mathit{ComputeAssignment} \\
\mathit{resource?} : \mathit{RESOURCE} \\
\mathit{resourcesAccess!} : \mathit{ROLE} \leftrightarrow \\
\qquad (\mathit{PERMISSION} \times (\mathit{ATOMIC_ACTION} \times \mathit{RESOURCE})) \\
\mathit{action_resource_set!} : \mathbb{F}(\mathit{ATOMIC_ACTION} \times \mathit{RESOURCE}) \\
\hline
\mathit{resourcesAccess!} = \{r : \mathrm{dom}\ \mathit{comp_Assignment};\ p : \mathit{permission}; \\
\qquad \mathit{atm} : \mathit{atm_action} \mid (r \mapsto (p, \mathit{atm}, \mathit{resource?})) \\
\qquad \in \mathit{concrete_Assignment} \bullet (r \mapsto (p, (\mathit{atm}, \mathit{resource?})))\} \\
\mathit{action_resource_set!} = \{x : \mathrm{ran}\ \mathit{resourcesAccess!} \bullet \mathit{second}(x)\}
\end{array}
$$

This operation also takes an input resource? of the type RESOURCE and computes the related roles and atomic actions of that resource; action_resource_set! ensures that only the atomic actions corresponding to the resources are retrieved. This schema is exemplified below: the input is resource MedicalRecords, and the result produced by Jaza is printed[1].

```
JAZA> ;EvaluateResourcesAccess
  Input resource? = MedicalRecords
  action\_resource\_set!=={(Read, MedicalRecords),(Write,
  MedicalRecords), (Delete, MedicalRecords),(Create, MedicalRecords)}
```

5.4 Permissions for Atomic Action and Role

The operation schema FindPermissions is intended to query the permissions for both a given atomic action and a role. This schema has two input parameters i.e., atm_action? and role? of the types ATOMIC_ACTION and ROLE, respectively. The predicate computes the set of permissions for the input role and the atomic action. As a result, perms! will return the set of all associated permissions for the input values. We need to give the atomic action along with the role as input, and it will return the permissions linked to them.

[1] Here and in the following we only show the relevant outputs.

$$
\begin{array}{l}
\textit{FindPermissions} \\
\hline
\Xi Sets;\ \Xi ComputeAssignment \\
atm_action? : ATOMIC_ACTION \\
role? : ROLE \\
perms! : ROLE \leftrightarrow \\
\qquad (PERMISSION \times ATOMIC_ACTION \times RESOURCE) \\
\hline
perms! = \{p : permission;\ rsrc : resource \mid \\
\qquad (role? \mapsto (p, atm_action?, rsrc)) \in concrete_Assignment \bullet \\
\qquad (role? \mapsto (p, atm_action?, rsrc))\}
\end{array}
$$

The following query is a result for the provided action Read and the role Doctor. Jaza tells us that there is one read permission associated to a doctor, named PatientRecord.

```
JAZA> ;FindPermissions
  Input atm\_action? = Read
  Input role? = Doctor
  perms!== {(Doctor, ("PatientRecord", Read, MedicalRecords))}
```

5.5 Finding Duplicate Roles

The schema FindDuplicateRoles allows us to search for duplicate roles. This query is useful to determine whether two roles have the same privileges in a secure system. This schema returns two roles, which are different but are associated with the same sets of atomic actions.

$$
\begin{array}{l}
\textit{FindDuplicateRoles} \\
\hline
\Xi Sets;\ \Xi ComputeAssignment \\
role1!, role2! : ROLE \\
aSet1!, aSet2! : \mathbb{F}\, ATOMIC_ACTION \\
\hline
role1! \in role \land role2! \in role \\
role1! \neq role2! \\
aSet1! = \{p : permission;\ a : ATOMIC_ACTION;\ rsrc : resource \mid \\
\qquad (role1! \mapsto (p, a, rsrc)) \in concrete_Assignment \bullet a\} \\
aSet2! = \{p : permission;\ a : ATOMIC_ACTION;\ rsrc : resource \mid \\
\qquad (role2! \mapsto (p, a, rsrc)) \in concrete_Assignment \bullet a\} \\
aSet1! = aSet2!
\end{array}
$$

The following query reports that Patient and EmergencyOfficer are duplicate roles, more precisely they have the permissions to perform the same actions.

```
Jaza ; FindDuplicateRoles
role1!==Patient, role2!==EmergencyOfficer
```

Availability of Data. Utilization of a particular service is handled by availability properties, which are particularly relevant in emergency situations. The availability properties offered by RBAC deal with granting permissions, which will ensure that a resource is available to a user. RBAC aims at avoiding undesirable states in which a user who is entitled to an access permission does not get it. To this end, we propose formal queries, which can be used to review RBAC-based policies. For example, it is significant to determine the minimum information about a patient that can be accessed by everyone. Thus, the availability of operations in our designed policy that could be used in such cases has to be checked.

5.6 Atomic Action Accessed by All

The operation schema AccessAll returns the atomic operations accessible by all roles of a system. The declaration part includes an output variable action!. The given predicate returns the atomic actions accessible by all roles.

AccessAll

$\Xi Sets;\ \Xi ComputeAssignment$
$action! : ATOMIC_ACTION$

$\forall r : role \bullet (\exists p : permission;\ rsrc : resource \bullet$
$\quad (r \mapsto (p, action!, rsrc)) \in concrete_Assignment)$

```
Jaza ;AccessAll
action!==Read
```

5.7 Atomic Action Access by Nobody

The operation schema AccessNobody returns the atomic action, which is completely inaccessible by all roles.

AccessNobody

$\Xi Sets;\ \Xi ComputeAssignment$
$action! : ATOMIC_ACTION$

$\forall r : role \bullet (\forall p : permission;\ rsrc : resource \bullet$
$\quad (r \mapsto (p, action!, rsrc)) \notin concrete_Assignment)$

It includes an output variable action!, which has the type of an atomic action. The predicate part checks for the inaccessible atomic actions.

```
JAZA> ;AccessNobody
  action!==Create
```

6 Separation of Duty Constraints

Separation of Duty (SoD) constraints are an optional construct of RBAC and are used to address conflicts of interest among roles, which consist of two categories, Static Separation of Duty (SSD) and Dynamic Separation of Duty (DSD).

The SSD takes care of conflicts of interest and ensures that a user does not take some conflicting roles even in different sessions. These constraints are specified over UA assignments as pairs of roles. UA is restricted during sessions. This ensures that if a user is assigned to a role, the user can never take the prohibited role. SSD can be applied not only to colluding users but also to groups, which are collections of users. Permissions can be associated with both users and groups.

DSD is the second kind of constraint offered by RBAC (Fig. 1). These constraints are intended to limit the permissions that are available to a user, whilst SSD constraints reduce the number of potential permissions that can be made available to a user. This is realized by placing constraints on the users that can be assigned to a set of roles. The main difference between SSD and DSD constraints lies in the context in which they are used. SSD are imposed on user's total permission space, but DSD restricts the users to activate the roles within or across a user's sessions. For example, a user Bob may have been assigned with two roles i.e., *Doctor* and *Patient*, but he may not exercise the permissions of both roles in the same session. In the RBAC model, a session is a traditional way of communicating information between a user and a system during a given time interval. Session management in RBAC deals with functions such as session creation for users including role activation/deactivation, enforcing constraints (e.g., DSD) on role activation. An obligatory part of DSD constraints is the use of sessions. In the following, we formally specify SoD constraints using the Z notation.

RoleAssignment

Sets
$conflicting_Roles : ROLE \leftrightarrow ROLE$
$role_Assignment : USER \leftrightarrow ROLE$

$\mathrm{dom}\ conflicting_Roles \subseteq role$
$\mathrm{ran}\ conflicting_Roles \subseteq role$
$\mathrm{dom}\ role_Assignment \subseteq user$
$\mathrm{ran}\ role_Assignment \subseteq role$
$\forall u : user \bullet (\forall i, j : role$
$\quad\mid ((u \mapsto i) \in role_Assignment) \wedge ((u \mapsto j) \in role_Assignment)$
$\quad\bullet ((i, j) \notin conflicting_Roles)))$

The declaration in RoleAssignment contains relations describing conflicting roles (conflicting_Roles) and for assigning roles to users (role_Assignment). They are defined as part of the security policy of a system. The first four constraints ensure that the relations to which the schema is applied are actually defined on

the roles and users from Sets (cf. Sect. 4.2). The last predicate specifies that any two roles assigned to a user are not from the conflicting roles set.

The subsequent schema SessionRoles formalizes DSD constraints. The schema includes one partial function session_User, because the users of a session have to be considered when checking the role assignment. The relation session_Role assigns roles to sessions, and like in the previous schema, conflicting_Roles_DSD describes the conflicting pairs of roles.

In the predicate part, the constraints for restricting domain and range (similarly to RoleAssignment) have been omitted. The first listed constraint states that whenever a user is assigned to a session with specific roles, the user should have these as pre-assigned roles. The last constraint specifies that the roles taken in one session should not be contained in the set of conflicting roles.

SessionRoles

$Sets;\ RoleAssignment$
$session_User : SESSION \nrightarrow USER$
$session_Role : ROLE \leftrightarrow SESSION$
$conflicting_Roles_DSD : ROLE \leftrightarrow ROLE$

$\vdots$
$\forall\, r : role \bullet (\forall\, s : session$
$\quad \bullet (r, s) \in session_Role \Rightarrow (session_User(s), r) \in role_Assignment)$
$\forall\, s : session \bullet (\forall\, i, j : role \mid ((i, s) \in session_Role) \wedge ((j, s) \in session_Role)$
$\quad \bullet ((i, j) \notin conflicting_Roles_DSD))$

In the security policy of Fig. 2 let us assume that a doctor is permitted to exercise two roles, i.e., a patient and a doctor. This can be regarded as a serious threat to the medical records where a patient, actually a doctor, compromises the information integrity, because a doctor may perform operations which a patient is not supposed to perform. However, such scenarios are avoidable by introducing an SSD constraint such that a doctor and a patient are specified as conflicting roles. However, this restricts the doctor who might be a patient at some point. In turn, as a solution, we can employ a DSD constraint, which enables exercising both roles but not in one session. Similarly, the role hierarchy (see Fig. 1) can be combined with SSD or DSD to avoid conflicting roles within the hierarchy.

7 Conclusions and Perspectives

This paper presents a formal approach to reviewing healthcare security policies. The proposed approach integrates the Z notation with security design models in order to assess access control rules of an EHR system. The applied idea follows the security-by-design principle and hence exhibits a strategy to cope with internal threats by investigating security properties such as integrity and confidentiality. The Jaza tool is applied to validate formal specifications. Note that

in our approach, the formally translated model (the initial state space) does not grow, and it avoids any further complex computations except using the queries to validate the model.

Abundant research literature can be found on how to translate graphical models to formal notations. Nonetheless, to reap out benefits from such formal translations, it is necessary to apply tools and techniques that facilitates easy validation and verification of formal models. Inspired by this, our approach takes such gaps into account. Also, the approach does not require mathematically skilled validation engineers for the following reasons: 1) working with only graphical models of security policies, and 2) automated translation of graphical models besides the reviewing queries. The approach is generic in a sense that it can be used to design and validate other secure information systems irrespective of a particular domain. The paper has only addressed the internal threats (i.e., from the system users). However, UML profiles such as UMLsec [Jür05] can be applied to model and verify systems against external threats.

Currently, the SoD constraints of RBAC are inherently restricted: For instance, a hospital may require an emergency officer to have four roles out of six, but SoD constraints can only be applied over a pair of roles. Our future work includes extending this toolset by overcoming such deficiencies as well as automating the query generation process to help building quality models in the healthcare domain. The tool's performance will also be evaluated using larger models with an extended and complete set of queries.

Acknowledgments. This work has been supported by the projects SAFEHR and GAVES funded by Macao Science and Technology Development Fund, and partly supported by the ANR Selkis Project under grant ANR-08-SEGI-018.

References

[AH07] Ahn, G.-J., Hu, H.: Towards realizing a formal RBAC model in real systems. In: Lotz, V., Thuraisingham, B.M. (eds.) Proceedings of the 12th ACM Symposium on Access Control Models and Technologies, SACMAT 2007, Sophia Antipolis, France, June 20-22, pp. 215–224. ACM (2007)

[AJ10] Appari, A., Johnson, M.E.: Information security and privacy in healthcare: current state of research. Int. J. of Internet and Enterprise Management 6(4), 279–314 (2010)

[AK06] Abdallah, A.E., Khayat, E.J.: Formal Z specifications of several flat role-based access control models. In: 30th Annual IEEE/NASA Software Engineering Workshop (SEW), pp. 282–292. IEEE CS (2006)

[BCDE09] Basin, D.A., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Information & Software Technology 51(5), 815–831 (2009)

[BL75] Bell, D., LaPadula, L.: Secure computer system: Unified exposition and multics interpretation. Technical report, MITRE Corp, Bedford (1975)

[Bos95] Boswell, A.: Specification and validation of a security policy model. IEEE Trans. Software Eng. 21(2), 63–68 (1995)

[Bow03] Bowen, J.: Formal Specification and Documentation using Z: A Case Study Approach. Thomson Publishing (2003)

[DOD85] DOD 5200.28-STD. Trusted computer system evaluation criteria. Technical report, United States Department of Defense (1985)

[DW96] Davies, J., Woodcock, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall (1996) ISBN 0-13-948472-8

[FSG+01] Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. 4(3), 224–274 (2001)

[Hal94] Hall, A.: Specifying and interpreting class hierarchies in Z. In: Bowen, J.P., Hall, J.A. (eds.) Z User Workshop, pp. 120–138. Springer (1994)

[HY06] Hasan, R., Yurcik, W.: A statistical analysis of disclosed storage security breaches. In: Proceedings of the 2006 ACM Workshop on Storage Security and Survivability, StorageSS 2006, Alexandria, VA, USA, October 30, pp. 1–8. ACM (2006)

[Jou09] Rubenstein, S.: Are your medical records at risk? Wall Street Journal (2009)

[Jür05] Jürjens, J.: Secure systems development with UML. Springer (2005)

[LBD02] Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-based modeling language for model-driven security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)

[LIM+11] Ledru, Y., Idani, A., Milhau, J., Qamar, N., Laleau, R., Richier, J.-L., Labiadh, M.-A.: Taking into account functional models in the validation of IS security policies. In: Salinesi, C., Pastor, O. (eds.) CAiSE Workshops 2011. LNBIP, vol. 83, pp. 592–606. Springer, Heidelberg (2011)

[LQI+11] Ledru, Y., Qamar, N., Idani, A., Richier, J.-L., Labiadh, M.-A.: Validation of security policies by the animation of Z specifications. In: Breu, R., Crampton, J., Lobo, J. (eds.) Proceedings of the 16th ACM Symposium on Access Control Models and Technologies, SACMAT 2011, Innsbruck, Austria, June 15-17, pp. 155–164. ACM (2011)

[MIL+11] Milhau, J., Idani, A., Laleau, R., Labiadh, M.-A., Ledru, Y., Frappier, M.: Combining UML, ASTD and B for the formal specification of an access control filter. Innov. Syst. Softw. Eng. 7, 303–313 (2011)

[MSGC07] Morimoto, S., Shigematsu, S., Goto, Y., Cheng, J.: Formal verification of security specifications with common criteria. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, pp. 1506–1512. ACM (2007)

[QLI11] Qamar, N., Ledru, Y., Idani, A.: Validation of security-design models using Z. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 259–274. Springer, Heidelberg (2011)

[SM02] Schaad, A., Moffett, J.D.: A lightweight approach to specification and analysis of role-based access control extensions. In: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies, pp. 13–22. ACM (2002)

[Spi92] Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall International Series in Computer Science (1992)

[TMB06] Teije, A., Marcos, M., Balser, M., et al.: Improving medical protocols by formal methods. Artif. Intell. Med. 36(3), 193–209 (2006)

[TRA+09] Toahchoodee, M., Ray, I., Anastasakis, K., Georg, G., Bordbar, B.: Ensuring spatio-temporal access control for real-world applications. In: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, pp. 13–22. ACM, New York (2009)

[YHHZ06] Yuan, C., He, Y., He, J., Zhou, Z.: A verifiable formal specification for RBAC model with constraints of separation of duty. In: Lipmaa, H., Yung, M., Lin, D. (eds.) Inscrypt 2006. LNCS, vol. 4318, pp. 196–210. Springer, Heidelberg (2006)

[ZWCJ02] Zao, J., Wee, H., Chu, J., Jackson, D.: RBAC schema verification using lightweight formal model and constraint analysis. Technical report, MIT, Cambridge (2002)