Automatic Generation of Security Compliant (Virtual) Model Views

Salvador Martínez[1], Alexis Fouche[1], Sébastien Gérard[1] and Jordi Cabot[2]

[1] CEA-LIST, Paris-Saclay, France
{salvador.martinez,alexis.fouche,sebastien.gerard}@cea.fr
[2] ICREA-UOC, Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. The increased adoption of model-driven engineering in collaborative development scenarios raises new security concerns such as confidentiality and integrity. In a collaborative setting, the model, or fragments of it, should only be accessed and manipulated by authorized parties. Otherwise, important knowledge could be unintentionally leaked or shared artifacts corrupted. In this paper we explore the introduction of access-control mechanisms for models. Our approach relies on the definition of a domain specific language tailored to the definition of access-control rules on models and on its enforcement thanks to the automatic generation of security compliant (virtual) views.

1 Introduction

The increased adoption of the model driven engineering (MDE) paradigm in complex collaborative scenarios introduces the need for effective confidentiality and integrity protection mechanisms at the modeling level. Indeed, in such scenarios, a given model (or type of model) may be shared among different stakeholders over possibly untrusted channels. This model, or fragments of it, should only be accessed and manipulated by authorized parties. Otherwise, the collaboration process could lead to the leak of important knowledge, likely triggering reputation and/or economical losses.

Access-control (AC) policies are often the mechanism of choice to implement the security requirements of confidentiality and integrity and thus, they constitute a pervasive mechanism in current information systems. However, while there exist standard access-control languages based on well-defined paradigms (e.g., Role-based access-control (RBAC) [4] and attribute-based access control (ABAC) [12]) the available AC frameworks can not be directly used in an MDE scenario as they either provide a coarse granularity that only allow for managing permissions at the file level (e.g., file systems rights), requiring the manual fragmentation of the model to enforce security, or do not take into account specificities of the modelware technical space such as the existence of metamodels and the conformance relation.

This makes it difficult to define and enforce policies that respect the *least privilege principle* that states that subjects must only have the rights they need to perform their assigned duties. Indeed, an effective access-control mechanism for models must 1) provide the means to define and enforce fine-grained access-control rules (i.e., the means

to control access to any part of a model, be it a class, an attribute, a relation or an operation); 2) protect both models and their metamodels since metamodel information is also valuable and should be protected; 3) be usable by modelers (which requires a language that uses familiar model concepts as language primitives) and 4) keep the consistency of the secured models, so that they can be viewed and manipulated by existing MDE tools with no adaptation.

To the best of our knowledge, a language satisfying all these constraints does not exist. While modeling has been intensively used as a means of including security and access-control concerns in the early phases of systems design and specification[7] [8], very few approaches are specially tailored to the protection of the models themselves , e.g.[3], and none of them consider the modification of a model's metamodel as a requirement and enforcement mechanism.

Thus, we have decided to build a new fine-grained access-control mechanism for models. More specifically, in this paper we present an approach composed of: 1) a role-based access-control language specially designed to work with models allowing for the specification of conditions at the M2 and M1 level and 2) an enforcement mechanism based on the automatic generation of security compliant (virtual) views that protects both the model and metamodel.

2 Concepts

In order to ease the discussion, we give a few notions on the concepts of access-control and model views.

-Access-Control: Figure 1 (a) shows the core concepts of access-control: Objects represent the passive resources that can be accessed within a system and that we may want to protect (files in an operation system, tables in a database,...). **Subjects** are the active entities in a system. They represent the actors to which the access to *Objects* is controlled. **Actions** are any kind of access to the *Objects* that may be performed by the *Subjects* in a given system. **Permissions** relate *Actions* with *Objects*. A permission is thus the right to perform a given *Action* (or set of actions) on a given *Object* (or set of objects). These permission are, in turn, granted to *Subjects*.

Nevertheless, directly assigning permissions to Subjects becomes unpractical when the user-base of the applications is large. Hence, in real applications, the definition of the permissions and its assignment is often performed by using the concepts of **Rule** and **Policy**. A rule is the assignment (or denial) of a permission to a given subject. Generally, access control rules have the form: $R_i : \{conditions\} \rightarrow \{decision\}$, where the sub-index *i* specifies the ordering of the rule, *decision* can be accept or deny and *conditions* is a set of rule matching attributes (e.g., hold roles). An access-control security policy is the set of permission assignments within a given information system, which is composed of a set of Rules. This policy constitutes a mere definition of the security requirements for the system, while the process of implementing the mechanisms to make the system follow the rules it defines is called enforcement.

-Model-Views: The concept of view is very common in the field of databases, and serves as a way to provide a certain perspective tailored for a specific type of use.

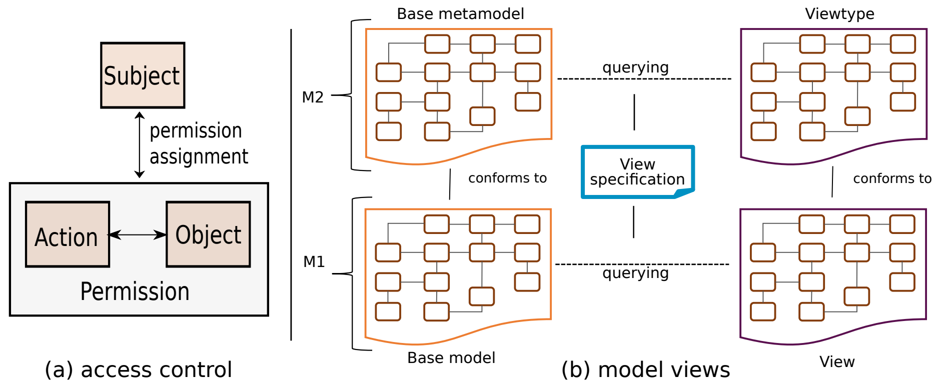

Fig. 1. Core Concepts

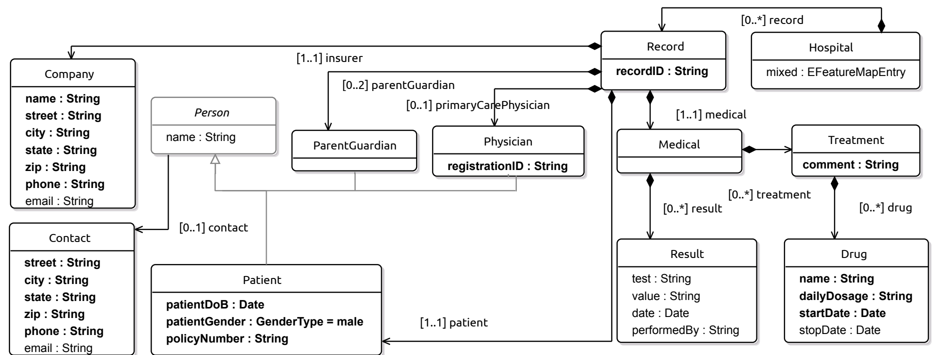

Fig. 2. Medical Record Running Example

As such, a database view may be considered as a security enforcement mechanism that filters and restricts the information a certain user can see from the database. There exists several solutions bringing the concept of views to the modelware realm using a number of different strategies. We base our work on the *virtual models* approach where views are not serialized but instead are the result of executing live queries on the original model. In particular, we adopt the terminology described in [2]. The main concepts (depicted in Figure 1 (b)) are: *Base meta-model*: a regular meta-model involved in the definition of a viewtype; *base model:* a regular model used as an input for building a view; *viewtype:* a metamodel which structure is defined through the specification of queries on one (or more) meta-model; *view:* a model conforming to a viewtype, and resulting from the querying of one (or more) base model.

3 Approach

This section introduces the main elements of our approach and a running example to illustrate them.

Running Example: We show in Figure 2 the metamodel of a medical record, the same example used in the XACML specification[10]. Medical *records*, identified by a *recordid* string, contain five different types of information: 1) information about the insurer of the patient represented by the *Company* metaclass; 2) Information about the *Patient*, including name and *Contact* information; 3) information about the parents of the patient represented by the *ParentGuardian* metaclass; 4) information about the prescribed treatments, represented by the *Medical* metaclass that aggregates data regarding the *Treatment* and the *Drugs* it uses and regarding the *Result* of medical visits; and finally 5) information about the *physician* assigned to that patient. Given this metamodel, there are different security scenarios we may want to consider: 1) Hiding part of the metamodel to a partner, e.g. if we are outsourcing the development of the "people" subsystem, we may want to hide to that partner the existence of classes to store medical information on treatments and drugs, 2) Restricting access to medical information based on the profile of the user, e.g. in a models@run.time scenario, we may want to block access to record objects except to physicians in charge of that patient. Note that scenario 1 involves defining a rule at the "type-level" while the rule for scenario 2 involves the "instance-level". We support both kinds of rules (and combinations of both).

Our Solution: Figure 3 summarizes our approach for providing access-control for models. The process starts with a security engineer that specifies the desired policy. This policy is written using a language specially tailored to define such modeling access control rules. This policy is then transformed to a viewtype specification. The viewtype specification is interpreted by a view engine in order to generate a filtered metamodel (viewtype) and eventually, a filtered model (view). These are the elements the end-user will obtain upon an access request. Note that different filtered artefacts (viewtypes and views) are generated for each accessing role or end-user.

Our approach is thus composed of two main building blocks, an AC language and a view generator in charge of the enforcement of the policy defined with such AC language. We provide tool support[3] for the AC language, its transformation towards a view specification, and the execution of such view specification.

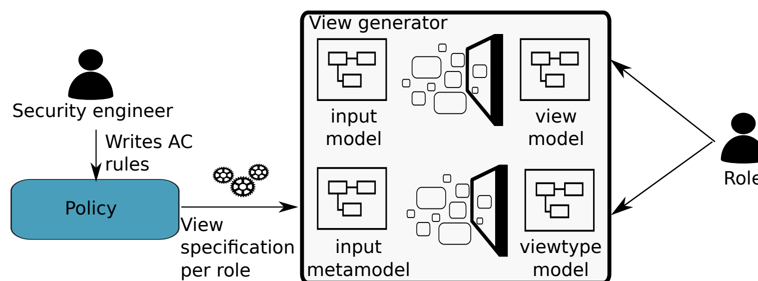

Fig. 3. Approach

[3] `https://gitlab.com/smartine/SecureModelViews`

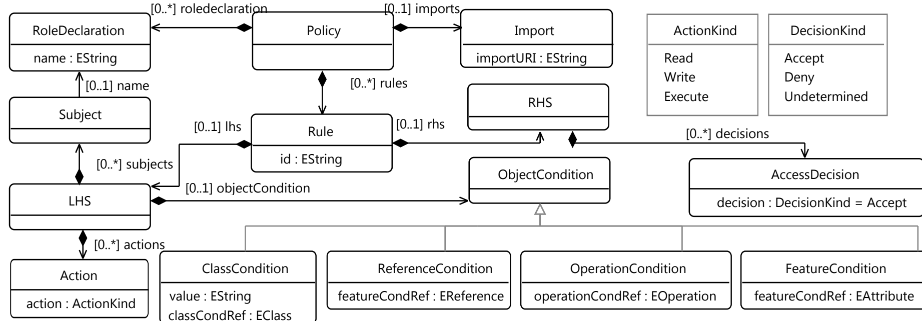

Fig. 4. Access-control Policy Metamodel

4 Access-Control Language for Models

The first step requires providing the means to define access-control rules for models. We do so by creating a domain specific policy language. This language allows security engineers to write rules based on the base metamodel (i.e. filtering access based on certain element types; the types themselves should not even be visible to the users) and/or its instances (i.e. preventing access to model elements with certain values). In the rest of this section we discuss the language's abstract syntax and its execution semantics and we provide a textual concrete syntax to ease its utilization.

4.1 Abstract Syntax

Figure 4 shows the conceptual schema of our RBAC-based policy language. It allows the definition of rules that associate roles to the permissions (or prohibitions) to perform actions on model elements. In this language, a *Policy* contains a number of access-control *Rules*. These rules are composed of a left-hand side and a right-hand side. The left-hand side is meant to be used to express a number of conditions for a given access-control rule to apply to a given access request. We provide our language with three specific condition elements:
1) *Subject* identified by its reference to a *RoleDeclaration* and representing the subject accessing the protected resource.
2) *Action* that represents the operation to be performed on the protected resource. The values of the *actionType* attribute of type *ModelAction* can be *Read*, *Write* or *Execute* (this action only applies to model operations). Note that we include the three CRUD operations *Create*, *Update* and *Delete* on the *Write* operation as the granularity of our language permits to obtain the same effects by the combination of *Write* and *Read* operations on the different model elements.
3) *ObjectCondition* that represents the resource to which the rule applies. Our language allows the definition of *ClassCondition* to express permissions that apply to a class, *AttributeCondition* *ReferenceConditions* that are meant to represent permissions on specific attributes and references and *OperationCondition* to represent permissions on model operations. Note that all *ObjectCondition* elements hold a reference to the metamodel element they refer to. Besides, the *ClassCondition* holds a *value* attribute

of type String meant to be a place holder for model queries to filter model elements w.r.t. complex conditions. Concrete implementations may link this query place holder to concrete query languages such as OCL [11]. Their evaluation is nevertheless delegated to the view framework in charge of generating the view models from a given view specification.

The right-hand side of rules is used to express the effect the application a *Rule* has by means of a *Decision*. *DecisionKind* lists the type of decisions that can be issued, namely *Allow* for granting a permission, *Deny* for a prohibition. Note that the *Policy* holds a *default* attribute of type *DefaultPolicy* used to provided a general decision when no rule applies. This policy language can be easily enhanced so that elements such as roles and actions have attributes as we have done in a previous work [9]. Finally, we consider user management issues (such as role hierarchies or role delegation) and policy constraints (such as Separation of Duty) out of the scope of this paper. Nevertheless, our language may be easily extended to support advanced concepts and integrate contributions where OCL constraints are used to impose constraints on the policy [1].

4.2 Concrete Syntax

In order to ease its use, we also define a textual concrete syntax for the Policy language. Access-control policies may easily become large and thus, a textual syntax would be easier to read and manipulate than a graphical or form-based one. Listing 1.1 shows an example of this concrete syntax. A policy with a default behaviour of *deny* is defined for our Medical Record example introduced in Section 2. The policy defines three roles, Physician, Patient and Clerk and then proceeds to define access-control rules for the Clerk role. Rules c1 to c4 grant read access to the Company, Hospital, Patient and Record elements. Rule c3 However restricts the granted access to Records with recordID greater than 100. c5 to c7 deny the access to the Physician, Medical and Parent elements. Finally, rule c8 forbids the access to the *patientGender* attribute of Patient.

As we can see, the access control policy mixes positive with negative permissions. This simplifies the propagation of permissions leading to policies that are more compact and easier to read. Also, it does not list all of the metamodel classes, attributes, references and operations (which would be tedious and error prone). For its correct interpretation we need to provide our language with a set of precise execution semantics. We do so in the following subsection.

Listing 1.1. Policy Example

```
import "platform//Test1/records.ecore"

DeclareRole Physician, Patient, Clerk

rule c1 (Clerk; Read; class records.Company)-> Accept
rule c2 (Clerk; Read; class records.Hospital)-> Accept
rule c3 (Clerk; Read; class records.Record
  WithValue =< "self.recordID > 100" >)-> Accept
rule c4 (Clerk; Read; class records.Patient)-> Accept
rule c5 (Clerk; Read; class records.Physician)-> Deny
rule c6 (Clerk; Read; class records.Medical)-> Deny
rule c7 (Clerk; Read; class records.Parent)-> Deny
rule c8 (Clerk; Read; att records.Patient.patientGender)-> Deny
```

4.3 Execution Semantics

In the general case, the calculation of the permissions for each metamodel and model element requires 1) to match an applicable access-control rule if such a rule exists and 2) to apply permissions propagation policies that may affect the evaluation of that rule. These two mechanisms work as follows:

Rule matching. As shown in Figure 4, each rule may list several *Role* and several *Action* elements. However, they only list one metamodel element (*metaclass*, *structuralfeature*, or *operation*). This limitation simplifies the process of matching applicable rules and, more importantly, prevents rule conflicts due to the intersection of rule conditions (and well-formedness rules defined at the policy language level already prevent the existence of two rules on the same metamodel element with an overlapping set of roles). An additional generic query language helps to write arbitrary conditions to precisely define the set of instances of the metamodel elements affected by the rule.

Permission Propagation. In order to clarify the interpretation of access-control policies, we propose a numbers of permission propagation *principles* that simplify the specification of such rules freeing the designers from manually defining in each rule the priorities in case of conflict:

- *SuperClass Propagation*. Permissions defined for a superclass are inherited by the subclasses if no other rule is defined for the subclasses.
- *Containment Relationship Propagation*. Permissions are propagated through the containment relationship as defined in the metamodel.
- *Containment Propagation*. Permissions on a Class are propagated to its contained elements (i.e., its attributes, references and operations).
- *Deny Overrides.* A rule denying a permission on a given model element is propagated to all the subtree of contained elements overriding any rule granting the permission that may be found on the subtree (including its contained attributes, references and operations). This guarantees that the containment hierarchy is preserved, a requirement to have valid models.
- *Default Propagation*. If no access rule is applicable to a given model element, and no permission is inherited from the previous propagation principles, the default policy applies;

As a result of applying the aforementioned execution semantics to the example in Listing 1.1 we will obtain the following list of permissions: Accept Classes: *Hospital, Record (with RecordId greater than 100), Company and Patient*; Deny Classes: *Physician, ParentGuardian, Medical, Treatment, Result, Drug and Contact*; Accept all Attributes and References from Accept Classes apart from patientGender and contact; Deny all other Attributes and References. Note that the rule C1 is redundant, as *Company* inherits the accept permission from *Record*. We detect this kind of anomalies and report them to the user during the transformation process we describe in Section 5.

5 Enforcement with Virtual Model Views

As stated in Section 3, the second building block of our approach is a model view generator. Indeed, we use *views* as an access-control enforcement mechanism. We adapt

the implementation of the virtual model view approach as described in EMFViews[2] for that purpose. A view specification corresponding to an access-control policy conforming to our PolicyDSL language is automatically obtained by the use of a model transformation. This transformation takes as input three models: an access-control policy; the metamodel referred by that policy; and a parameters model indicating for which role the view is to be generated. It produces as output a view definition.

Due to space limitations we do not show here the transformation nor the View definition and Parameter metamodels. They are available in the project website together with a demo showing a view automatically derived from the policy in Listing 1.1.

The biggest advantage of enforcing access-control through the use of virtual model views resides in 1) its capacity to modify the metamodel of the model to be protected thanks to the generation of a specific viewtype for the view; 2) the elimination of the synchronization issues that would appear otherwise between the original model and the filtered parts (synchronization between the views and the original model is also supported at the individual attribute level, while more complex updates fall under the limits of the well-known view update challenge[5]).

As future work we intend to support other modelling artifacts, like OCL queries and model transformations that should be adapted when producing a view so that they continue to be executable and meaningful in the *secure* context. From a tooling perspective, we will complete the integration of the components described into the open source Papyrus UML environment [6].

References

1. A. Ben Fadhel, D. Bianculli, and L. Briand. GemRBAC-DSL:a high-level specification language for role-based access control policies. In *SACMAT'16*, pages 179–190. ACM, 2016.
2. H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot. EMF views: A view mechanism for integrating heterogeneous models. In *ER'15*, pages 317–325. Springer, 2015.
3. C. Debreceni, G. Bergmann, I. Ráth, and D. Varró. Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations. *SOSYM*, pages 1–33, 2017.
4. D. Ferraiolo, J. Cugini, and D. R. Kuhn. Role-based access control (RBAC): Features and motivations. In *ACSAC*, pages 241–48, 1995.
5. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TOPLAS*, 29(3):17, 2007.
6. S. Gérard et al. Papyrus uml. *URL: http://www. papyrusuml. org*, 8, 2012.
7. J. Jürjens. UMLsec: Extending UML for secure systems development. In *"UML'02"*, pages 412–425. Springer, 2002.
8. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *"UML'02"*, pages 426–441. Springer, 2002.
9. S. Martínez, J. García, and J. Cabot. Runtime support for rule-based access-control evaluation through model-transformation. In *SLE'16*, pages 57–69. ACM, 2016.
10. E. Rissanen et al. extensible access control markup language (XACML) 3.0, 2013.
11. O. Uml. 2.0 OCL specification. *OMG Adopted Specification (ptc/03-10-14)*, 2003.
12. E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *ICWS'05*. IEEE, 2005.