A Metamodel-Based Approach for Analyzing Security-Design Models

David Basin Jürgen Doser
Information Security Group
ETH Zürich
{basin,doserj}@inf.ethz.ch

Manuel Clavel Marina Egea
Computer Science Department
U. Complutense, Madrid
clavel@sip.ucm.es
marina_egea@fdi.ucm.es

ABSTRACT

Security-design models are models that combine design specifications for distributed systems with specifications of their security policies. We have previously proposed an expressive UML-based language for constructing and transforming security-design models. Here we show how the same framework can be used to analyze these models: queries about properties of the security policy modeled are expressed as formulas in UML's Object Constraint Language and evaluated over the metamodel of the security-design language. We show how this can be done in a semantically precise and meaningful way and demonstrate, through examples, that this approach can be used to formalize and check nontrivial security properties of security-design models. The approach and examples presented have all been implemented and checked in the SecureMOVA tool.

1. INTRODUCTION

Model driven development [10] holds the promise of reducing system development time and improving the quality of the resulting products. Recent investigations [2, 7, 8, 9] have shown that security can be integrated into system design models and that the resulting *security-design models* can be used as a basis for generating systems along with their security infrastructures. Moreover, when the models have a formal semantics, they can be reasoned about: one can query properties of models and understand potentially subtle consequences of the policies they define.

In our previous work [2], we showed how to systematically combine different design models with a security modeling language closely related to that of Role Based Access Control (RBAC). The design models were used to formalize different system views using UML notation. The security modeling language, called SecureUML, was used to formalize authorization restrictions in a UML-based language related to RBAC. Moreover, we gave SecureUML and its combination with different design languages a formal semantics and described translators that automatically generate distributed, middleware-based systems with complete, configured, access control infrastructure from security-design models.

Our focus in this paper is on formalizing and automatically analyzing security properties of security-design models formalized using SecureUML. In our setting, security-design models constitute formal objects with both a notation (or concrete syntax) and an abstract syntax. Security models themselves are described by a metamodel that formalize the structure of well-formed models. We show that in this setting, security properties of security-design models can be expressed as formulas in OCL [11], the Object Constraint Language of UML, and evaluated over instances of the metamodel. The result is an expressive language for formalizing queries, which utilizes the entire vocabulary present in the UML metamodel defining security-design models. We may formalize queries in this language that ask questions about the relationships between users, roles, permissions, actions, and even system states. An example of a typical query (taken from Section 6) is: are there two roles such that one includes the set of actions of the others, but the roles are not related in the role hierarchy? Such queries can be answered by evaluating OCL expressions over the UML metamodel. This approach has been implemented, and the examples checked, in the MOVA tool [5], a UML modeling tool built on top of the Maude system [4].

The idea of formulating OCL queries over role-based access control policies is not new. Our work is inspired by [1, 13], who first explored the use of OCL for querying RBAC policies, and we make comparisons in Section 8. Moreover, OCL is the natural choice for querying UML models. It is part of the UML standard [12] and expressions written in OCL can be used to constrain and query UML models. Given this previous work, we see our contributions as follows. First, we clarify the metatheory required to make query evaluation formally well-defined. This requires, in particular, precise definitions of both the metamodel of the modeling language and the mapping from object models to the corresponding instances of this metamodel. Second, we show the feasibility of this approach and illustrate some of its key aspects on a nontrivial example: a security-design modeling language from [2] that combines SecureUML and a component modeling language named ComponentUML. Finally, we provide evidence that OCL expressions, evaluated in the context of such a metamodel, can be used to formalize and check non-trivial security properties of security-design models. The approach presented here has been implemented and tested in the SecureMOVA tool and we give examples

of its use.

Outline. In Section 2, we describe the methodology underlying our general approach and, in Section 3, we provide background material on OCL. In Section 4, we describe SecureUML and ComponentUML, the security and design modeling languages we use. Afterwards, in Section 5, we describe the semantics of the security modeling language using OCL constraints on the metamodel. In Section 6, we give a number of examples, which illustrate how one can formalize and analyze different kinds of authorization decisions using OCL. Then, in Section 7, we present SecureMOVA, a security-design modeling tool whose implementation is directly based on our metamodel-based approach for analyzing security-design models. We conclude in Section 8 with a discussion of related and future work.

2. GENERAL APPROACH

Background: models, metamodels, and meaning. A *modeling language* provides a vocabulary (concepts and relations) for building models, as well as a notation to graphically depict the models as diagrams. Diagrams have to conform with the *metamodel* of the modeling language. A metamodel is a diagram whose graphical elements formalize the concepts and relations provided by the (object) modeling languages, and whose invariants, usually written in OCL, specify additional well-formedness constraints on models. The precise definition of *well-formed* diagrams is based on the underlying *mapping* from diagrams (or *graphical models*) to instances of the metamodel (or *abstract models*): well-formed diagrams are those that are mapped to instances of the metamodel that satisfy the metamodel's invariants.

Some modeling languages explain the *meaning* of the diagrams using natural language. In this situation, analyzing the models represented by the diagrams can only be done informally and no rigorous tool support can be expected. Other modeling languages explain the meaning of the diagrams using a *formal semantics*: that is, they define an interpretation function that associates mathematical structures to the well-formed diagrams, or, more precisely, to the instances of the metamodel that correspond to well-formed diagrams. In this case, properties of the models represented by the diagrams can be formally proven, possibly with the assistance of automated tools. In the following, let M be a graphical model (for a modeling language $\mathcal{M}$), $\overline{M}$ be the corresponding abstract model, and $[\overline{M}]$ be the mathematical structure associated to the abstract model by the interpretation function $[.]$.

Problem statement: rigorously analyzing security models. Given a language with a formal semantics, one can reason about models by reasoning about their semantics. That is, a security model M has a property P (where P is expressed in some logical language) if and only if $[\overline{M}] \models P$. While this approach is standard, it either requires deductive machinery for reasoning about the semantics of models (i.e., a semantic embedding [3] and deduction within the relevant semantic domains) or an appropriate programming logic for reasoning at the level of the models. These are strong requirements and a hurdle for many practical applications. Hence, the question we address is whether there are other ways of formally analyzing security policies modeled by M, but in a more familiar setting.

Approach taken. Our approach for analyzing properties of (security-design) models M reduces deduction to evaluation: we formalize the desired properties as OCL queries and evaluate these queries over instances $\overline{M}$ of the metamodel. Observe that these queries are formulated over the abstract models, not the (graphical) models that the modeler sees and works with. Hence, for the results to be meaningful, we require that the mapping relating graphical models to abstract models, along with the interpretation function $[.]$, correctly interacts with evaluation of OCL expressions. The precise requirements are defined below. If this mapping is not explicitly given, or the requirements are not satisfied, the validity of the results returned may be open, or even wrong (for examples, see the related work section).

Overall, our approach has a number of advantages over more traditional deductive approaches. First, OCL is a formal language defined as a standard add-on to UML. Hence, as noted in [15], "it should be easily read and written by all practitioners of object technology and by their customers, i.e., people who are not mathematicians or computer scientist". Second, many tools support the language in different ways. In particular, there are tools that can automatically evaluate OCL expressions. The limitations are also clear: there may be interesting properties that cannot be naturally expressed using OCL or that cannot be proved by simply evaluating OCL expression over the metamodel.

Correctness. Here we expand on the requirements of our approach, in particular how OCL query evaluation must related to the semantics of the modeling language. Let f be a function on the semantic domain and let exp_f be an expression intended to formalize f in OCL. We require the following diagram to commute:

$$\begin{array}{ccccc} \text{graphical Model} & & \text{abstract Model} & & \text{semantic Domain} \\ M & \mapsto & \overline{M} & \mapsto & [\overline{M}] \\ & & \downarrow & & \downarrow \\ & & ev(exp_f, \overline{M}) & \mapsto & f([\overline{M}]) \end{array}$$

In this diagram, the downwards arrow on the left side denotes the evaluation of the OCL expression exp_f (the result of which, denoted by the function $ev(\cdot,\cdot)$, constitutes another abstract model). The downwards arrow on the right side corresponds to the evaluation of the function f in the semantic domain. The requirement says that the OCL expression exp_f can be used to analyze the behavior of f if and only if $[ev(exp_f, \overline{M})] = f([\overline{M}])$. Roughly speaking, this means that an OCL expression can be correctly used for checking a property P if and only if, for arbitrary models M, the result of evaluating this expression over $\overline{M}$ corresponds to the value of the property P in $[\overline{M}]$.

Rigorously proving this correspondence requires detailed metareasoning that involves both the semantics of the underlying formal system, the (formal) semantics of OCL, and the translation scheme from terms in the semantic domain to OCL expressions. This is a large undertaking and outside the scope of this paper. In many practical cases however, one may settle for the next best thing: it may be sufficient

to have a careful understanding of the metamodel of the modeling languages and, in particular, its invariants, and of the underlying mapping from models to the corresponding instances of the metamodel. Note that, as explained above, this is already a necessary condition for stating meaningful OCL expressions on models in the first place.

3. OCL

As we have just explained, our general approach for analyzing security-design models is to formalize their properties as OCL expressions and to evaluate these expressions over instances of the metamodel. Before illustrating this approach with the security-design language that results from combining SecureUML and ComponentUML, we first briefly summarize relevant aspects of OCL.

The Object Constraint Language (OCL) [11] is a textual, typed language, with an object-oriented notation, for writing constraints within (or queries on) UML models. The language includes predefined types like Boolean, Integer and String, with standard operations like not and or, + and ∗, and substr and concat. For example, 2 + 5 and not(2 + 5 = 6) are OCL expressions of type Integer and Boolean, respectively. The language also provides mechanisms for generating collection types from more basic types, with a rich set of operations like union, includes, or size, with the expected meaning. For example, Set(Integer) is the type for the sets of integers; Set{1, 4, 6}−>union(Set{3}) is an expression of type Set(Integer) that denotes the union of the sets $\{1, 4, 6\}$ and $\{3\}$. Iterator operators like forAll, select, or collect, are operations on collection types. Each takes an OCL expression as an argument and specifies an operation computed over the elements of a collection. For example, Set{1, 4, 6}−>forAll(i|i > 7) is an expression of type Boolean that tests the property of being greater than 7 on each element of the set $\{1, 4, 6\}$.

The expressiveness of the language comes from the fact that OCL is really an open (parametric) language. Expressions are written with reference to a UML model, using the types and vocabulary provided by the model. The new types correspond to the classes in the model, and the new vocabulary correspond to the properties (attributes, roles, and operations) declared for these classes. For example, consider a class diagram M containing a class A. Suppose too that the class A has an attribute x of type String. Now, x can appear in OCL expressions, using dot notation: for an object o of the class A, the expression $o.x$ denotes the value of its attribute x. OCL also provides access to the value of some properties of the classes themselves using the dot notation: E.g., the expression A.allInstances() denotes the set of all instantiated objects of the class A.

4. THE SECUREUML + COMPONENTUML LANGUAGE

4.1 The SecureUML + ComponentUML Metamodel

SecureUML. SecureUML is a modeling language for formalizing access control requirements that is based on RBAC [6]. In RBAC one specifies access control policies in terms of static role assignments, however, it is not easily possible to specify policies that depend on dynamic properties of the system state. SecureUML extends RBAC with *authorization constraints* to overcome this limitation. It formalizes access control decisions that depend on two kinds of information:

- Declarative access control decisions that depend on static information, namely the assignments of users and permissions to roles, which we designate as a *RBAC configuration*.
- Programmatic access control decisions that depend on dynamic information, namely the satisfaction of authorization constraints in the current system state.

SecureUML provides a language for specifying access control policies for actions on protected resources. However, it leaves open what the protected resources are and which actions they offer to clients. These depend on the primitives for constructing models in the system design modeling language.

A SecureUML dialect specifies how the modeling primitives of SecureUML are integrated with the primitives of the design modeling language in a way that allows the direct annotation of model elements with access control information. Hence it provides the missing vocabulary to formulate security policies involving these resources by defining:

- the model element types of the system design modeling language that represent protected resources;
- the actions these resources types offer and hierarchies classifying these actions; and
- the default access control policy for actions where no explicit permissions is defined (i.e., whether access is allowed or denied by default),

ComponentUML. The ComponentUML language that we consider in this paper, is a simple language for modeling component-based systems. Essentially, it provides a subset of UML class models: *Entities* can be related by *Associations* and have *Attributes* and/or *Methods*.

The dialect definition then specifies the following.

- *Entities*, as well as their *Attributes*, *Methods*, and *AssociationEnds* (but not *Associations* as such) are protected resources.
- the actions offered by these resources are:

Resource	Actions
Entity	create, read, update, delete, full access
Attribute	read, update, full access
Method	execute
Association end	read, update, full access

The dialect definition also specifies the following action hierarchy.

EntityFullAccess: create, read, update, and delete of the entity.

EntityRead: read for all attributes and association ends of the entity, and execute for all side-effect free methods of the entity.

EntityUpdate: update for all attributes of the entity, update for all association ends of the entity, and execute for all non-side-effect free methods of the entity.

AttributeFullAccess: read and update of the attribute.

AssociationEndFullAccess: read and update of the association end.

- The default access control policy is *allow*.

Abstract Syntax and Metamodel. We define the abstract syntax of this modeling language following the standard approach taken in MDA, namely by giving a metamodel for it. Figures 4.1 and 4.2 present the abstract syntax of SecureUML and the ComponentUML dialect, respectively. We also need to add constraints to this metamodel in order to precisely define the well-formed models of SecureUML + ComponentUML. These constraints are presented in Appendix A.

4.2 The SecureUML + ComponentUML Models

SecureUML. SecureUML's concrete syntax is defined in [2] by a UML profile that formalizes the modeling notation of SecureUML using stereotypes and tagged values. This profile does not define an encoding for all SecureUML elements. For example, the notation for defining *resources* is left open and must be defined by the dialect. A role is represented by a UML class with the stereotype "Role" and an inheritance relationship between two roles is defined using a UML generalization relationship. The role referenced by the arrowhead of the generalization relationship is considered to be the superrole of the role referenced by the tail. A permission, along with its relations to roles and actions, is defined in a single UML model element, namely an association class with the stereotype "Permission". The association class connects a role with a UML class representing a protected resource, which is designated as the *root resource* of the permission. The actions that such a permission refers to may be actions on the root resource or on subresources of the root resource. Each attribute of the association class represents the assignment of an action to the permission, where the action is identified by the name and the type of the attribute. Stereotypes for these permission attributes specify how the attribute is mapped to an action, and are defined as part of the dialect. The authorization constraint expressions are attached to the permissions' association classes.

ComponentUML. ComponentUML uses a UML-based notation, where entities are represented by UML classes with the stereotype "Entity". Every method, attribute, or association end owned by such a class is automatically considered to be a method, attribute, or association end of the entity, so no further stereotypes are necessary.

Dialect Definition. The stereotypes "entityaction", "attributeaction", "methodaction", "associationendaction" respectively specify that a permission attribute refers to an action on an entity, attribute, method, or association end. The name of the permission attribute specifies the name of the attribute, method, or association end targeted by this permission. The type of the permission attribute specifies the action (e.g., read, update, or full access) that is permitted by this permission.

Example. As a running example, [2] considers a simplified system for administrating meetings. In Section 7 we will use this example to illustrate how one can mechanize the analysis of security policies using a tool implementing our metamodel-based approach for analyzing security-design models. In this example, the system should maintain a list of users and records of meetings. A meeting has an owner, a list of participants, a time, and a place. Users may carry out standard operations on meetings, such as creating, reading, editing, and deleting them. A user may also cancel a meeting, which deletes the meeting and notifies all participants by email. The system should obey the following (here informally given) security policy:

- All users of the system are allowed to create new meetings and read all meeting entries.
- Only the owner of a meeting is allowed to change meeting data and cancel or delete the meeting.
- A supervisor is allowed to cancel any meeting.
- A system administrator is allowed to read meeting data.

Figure 4.2 formalizes the above security policy using the concrete syntax for SecureUML + ComponentUML.

Mapping From Models to Metamodel Instances. Tables 1 and 2 in Appendix B present the mapping between SecureUML + ComponentUML models and their corresponding instances of the metamodel. Recall that, in our approach, the specification of security properties using OCL directly depends on this mapping since the expressions formalizing the properties will not be evaluated over the models, but over the corresponding instances of the metamodel.

To a large extent, this mapping is straightforward: UML model elements with appropriate stereotypes are mapped to instances of the corresponding metamodel elements and links between these metamodel instances are directly given by corresponding associations between the UML model elements. Importantly, however, in some cases this mapping is less straightforward. In particular, in cases where the concrete syntax provides convenient "syntactic sugar" to the modeler. We list below some examples of such subtleties. Let M be a model, then $\overline{M}$ contains (among others) the following elements:

- "Default" objects of type Role, AuthorizationConstraint, and Permission, which do not correspond to roles, authorization constraints, or permissions depicted in M.
- Objects of subtypes of Action which correspond to the actions offered by the resources, even if they are not mentioned in the attributes of the permissions depicted in M.
- Links between the "default" objects of type Permission, Role, and AuthorizationConstraint, and between the "default" object of type Permission and the objects of subtypes of Action. These links formalize the default access control policy defined in SecureUML + ComponentUML.

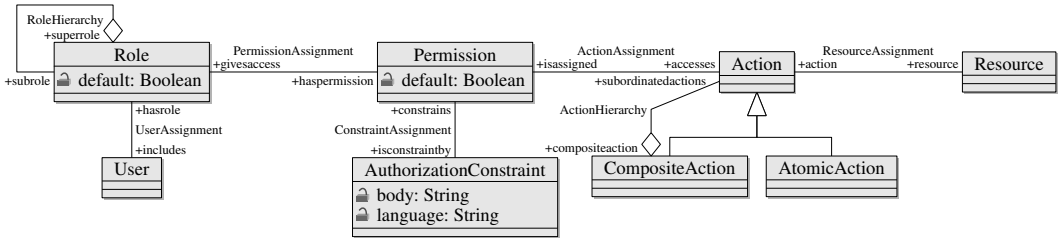

Figure 1: SecureUML Metamodel.

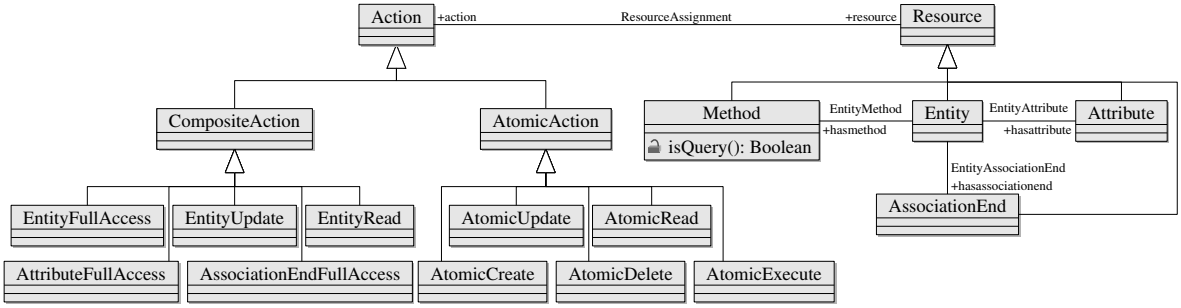

Figure 2: ComponentUML Dialect Metamodel.

- Links between the objects of subtypes of Action. These links formalize the hierarchy of actions defined in SecureUML + ComponentUML.

5. ANALYZING SECUREUML + COMPONENTUML MODELS

In this section, we define OCL operators over the metamodel of SecureUML + ComponentUML that formalize different aspects of the access control information contained in the models. We will use these as part of an OCL-based language for analyzing access control decisions that depend on static information, namely the assignment of users and permissions to roles.[1] The approach we take not only allows us to formalize desired properties of models, but also to automatically analyze models by evaluating the corresponding OCL expressions over the instances of the metamodel that corresponds to the models.

5.1 Semantics

We recall here the semantics of SecureUML + ComponentUML models [2], with respect to which we claim that our OCL-operations correctly capture access control information. Let $\Sigma_{RBAC} = (\mathcal{S}_{RBAC}, \leq_{RBAC}, \mathcal{F}_{RBAC}, \mathcal{P}_{RBAC})$ be an order-sorted signature that defines the type of structures specifying role-based access control configurations. Here $\mathcal{S}_{RBAC}$ is a set of sorts, $\leq_{RBAC}$ is a partial order on $\mathcal{S}_{RBAC}$, $\mathcal{F}_{RBAC}$ is a sorted set of function symbols, and $\mathcal{P}_{RBAC}$ is a sorted set of predicate symbols. In detail, let

$$\mathcal{S}_{RBAC} = \{Users, Roles, Permissions, AtomicActions, Actions\},$$

where $AtomicActions \leq Actions$,

$$\mathcal{F}_{RBAC} = \emptyset\,,$$

$$\mathcal{P}_{RBAC} = \left\{ \begin{array}{lcl} \leq_{Roles} & : & Roles \times Roles\,, \\ \leq_{Actions} & : & Actions \times Actions\,, \\ UA & : & Users \times Roles\,, \\ PA & : & Roles \times Permissions \\ AA & : & Permissions \times Actions \end{array} \right\}$$

Given a SecureUML + ComponentUML model M, one defines a Σ_{RBAC}-structure $\Im_{RBAC}$ in the obvious way: the sets *Users*, *Roles*, *Permissions*, *AtomicActions*, and *Actions* each contain entries for every model element in $\overline{M}$ of the corresponding metamodel types User, Role, Permission, AtomicAction, and Action. Also, the relations UA, PA, and AA contain tuples for each instance of the associations UserAsssignment, PermissionAssignment, ActionAssignment.

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any u in *Users* and r in *Roles*, $\Im_{RBAC} \models UA(u, r)$ if and only if

$$u.\mathsf{hasrole}{->}\mathsf{includes}(r)$$

evaluates to true in $\overline{M}$.

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model in M. Then, for any r in *Roles* and p in *Permissions*, $\Im_{RBAC} \models PA(r, p)$ if and only if

$$r.\mathsf{haspermission}{->}\mathsf{includes}(p)$$

evaluates to true in $\overline{M}$.

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any p in *Permissions* and a in *Actions*, $\Im_{RBAC} \models AA(p, a)$ if and only if

$$p.\mathsf{accesses}{->}\mathsf{includes}(a)$$

evaluates to true in $\overline{M}$.

5.2 Analysis Operations

In this section, we introduce a collection of OCL *query operations* that are useful for analyzing security properties of security-design models formalized using SecureUML +

[1] Programmatic access control decisions that depend on dynamic information, namely the satisfaction of OCL authorization constraints in concrete system states, can be then analyzed using standard OCL evaluators.

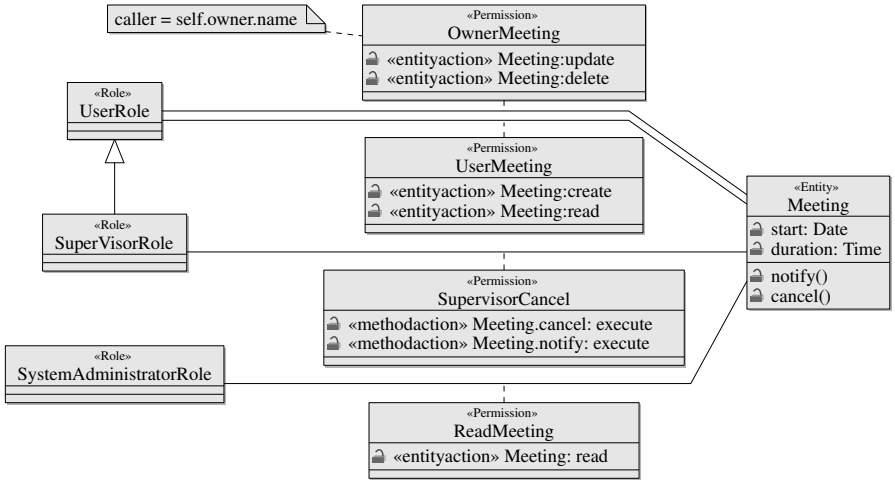

Figure 3: Example Security Policy.

ComponentUML. The correctness of these operations, with respect to the formal semantics of the language, is stated in the associated remarks.

The relation $\geq_{Roles}$ in the Σ_{RBAC}-structure $\Im_{RBAC}$ defined by a model M is given by the reflexive closure of the association RoleHierarchy on Role in $\overline{M}$; we write subroles (role with additional privileges) on the left (larger) side of the $\geq$-symbol.

superrolePlus. This returns the collection of roles (directly or indirectly) *above* a given role in the role hierarchy.

```
context Role::superrolePlus():Set(Role) body:
  self.superrolePlusOnSet(self.superrole)

context Role::superrolePlusOnSet(rs:Set(Role)):Set(Role)
body: if rs->collect(r1|r1.superrole)
          ->exists(r|rs->excludes(r))
      then self.superrolePlusOnSet(
          rs->union(rs->collect(r1|r1.superrole)->asSet()))
      else rs->including(self)
      endif
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any r_1, r_2 in *Roles*, $\Im_{RBAC} \models r_1 \geq r_2$ if and only if

$$r_2.\mathsf{superrolePlus}()\text{->}\mathsf{includes}(r_1)$$

evaluates to true in $\overline{M}$.

subrolePlus. This returns the collection of roles (directly or indirectly) *below* a given role in the role hierarchy.

```
context Role::subrolePlus():Set(Role) body:
    self.subrolePlusOnSet(self.subrole)

context Role::subrolePlusOnSet(rs:Set(Role)):Set(Role)
body: if rs->collect(r1|r1.subrole)
         ->exists(r|rs->excludes(r))
      then self.subrolePlusOnSet(
          rs->union(rs->collect(r1|r1.subrole)->asSet()))
      else rs->including(self)
      endif
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any r_1, r_2 in *Roles*, $\Im_{RBAC} \models r_1 \geq r_2$ if and only if

$$r_1.\mathsf{subrolePlus}()\text{->}\mathsf{includes}(r_2)$$

evaluates to true in $\overline{M}$.

allPermissions. This returns the collection of permissions (directly or indirectly) assigned to a role.

```
context Role::allPermissions():Set(Permission) body:
    self.superrolePlus().haspermission->asSet()
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Let $\phi_{Permission}(r_1, p)$ be the following formula:

$$\phi_{Permission}(r_1, p) \quad = \quad \exists r_2 \in \mathit{Roles}.\ r_2 \geq_{Roles} r_1 \wedge PA(r_2, p)\,,$$

with variables r_1 and r_2 of sort *Roles* and p of sort *Permission*. Then, for any r in *Roles* and p in *Permissions*, $\Im_{RBAC} \models \phi_{Permission}(r, p)$ if and only if

$$r.\mathsf{allPermissions}()\text{->}\mathsf{includes}(p)$$

evaluates to true in $\overline{M}$.

allRoles. This returns the collection of roles that are (directly or indirectly) assigned a permission.

```
context Permission::allRoles():Set(Role) body:
    self.givesaccess.subrolePlus()->asSet()
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any p in *Permissions* and r in *Roles*, $\Im_{RBAC} \models \phi_{Permission}(r, p)$ if and only if

$$p.\mathsf{allRoles}()\text{->}\mathsf{includes}(r)$$

evaluates to true in $\overline{M}$.

The relation $\geq_{Actions}$ in the Σ_{RBAC}-structure $\Im_{RBAC}$ defined by a model M is given by the reflexive closure of the composition hierarchy on actions, defined by the association ActionHierarchy in $\overline{M}$. We write $a_1 \geq_{Actions} a_2$, if a_2 is a subordinated action of a_1.

subactionPlus. This returns the collection of actions (directly or indirectly) subordinated to an action.

```
context Action::subactionPlus():Set(Action) body:
    if self.oclIsKindOf(AtomicAction)
    then Set{self}
    else self.oclAsType(CompositeAction)
         .subordinatedactions.subactionPlus()
    endif
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any a_1, a_2 in *Actions*, $\Im_{RBAC} \models a_1 \geq a_2$ if and only if

$$a_1.\mathsf{subactionPlus()}{->}\mathsf{includes}(a_2)$$

evaluates to true in $\overline{M}$.

compactionPlus. This returns the collection of actions to which an action is (directly or indirectly) subordinated.

```
context Action::compactionPlus():Set(Action) body:
    if self.compositeaction->isEmpty()
    then Set{self}
    else self.compositeaction.compactionPlus()
         ->including(self)
    endif
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any a_1, a_2 in *Actions*, $\Im_{RBAC} \models a_1 \geq a_2$ if and only if

$$a_2.\mathsf{compactionPlus()}{->}\mathsf{includes}(a_1)$$

evaluates to true in $\overline{M}$.

allActions. This returns the collection of actions whose access is (directly or indirectly) granted by a permission.

```
context Permission::allActions():Set(Action) body:
    self.accesses.subactionPlus()->asSet()
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Let $\phi_{Actions}(p, a_1)$ be the following formula:

$$\phi_{Actions}(p, a) \quad = \quad \exists a_2 \in \mathit{Actions}.\ a_2 \geq_{Actions} a_1 \wedge AA(p, a_2)\,,$$

with variables a_1 and a_2 of sort *Actions* and p of sort *Permission*. Then, for any p in *Permissions* and any a in *Actions*, $\Im_{RBAC} \models \phi_{Action}(p, a)$ if and only if

$$p.\mathsf{allActions}{->}\mathsf{includes}(a)$$

evaluates to true in $\overline{M}$.

allAssignedPermissions. This returns the collection of permissions that (directly or indirectly) grant access to an action.

```
context Action::allAssignedPermissions():Set(Permission)
body: self.compactionPlus().isassigned->asSet()
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any p in *Permissions* and any a in *Actions*, $\Im_{RBAC} \models \phi_{Action}(p, a)$ if and only if

$$a.\mathsf{allAssignedPermisssions()}{->}\mathsf{includes}(p)$$

evaluates to true in $\overline{M}$.

allAllowedActions. This returns the collection of actions that are permitted to a user, subject to the satisfaction of the associated constraints in each concrete scenario.

```
context User::allAllowedActions():Set(Action) body:
    self.hasrole.allPermissions().allActions()->asSet()
```

Remark: Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Let $\phi_{RBAC}(u, a)$ be the following formula:

$$\begin{aligned}\phi_{RBAC}(u, a) \quad = \quad & \exists r_1, r_2 \in \mathit{Roles}.\\ & \exists p \in \mathit{Permissions}.\ \exists a' \in \mathit{Actions}.\\ & UA(u, r_1) \wedge r_1 \geq_{Roles} r_2 \wedge PA(r_2, p)\\ & \wedge AA(p, a') \wedge a' \geq_{Actions} a\,,\end{aligned}$$

with variables u of sort *Users* and a of sort *Actions*. Then, for any u in *Users* and any a in *Actions*, $\Im_{RBAC} \models \phi_{RBAC}(u, a)$ if and only if

$$u.\mathsf{allAllowedActions()}{->}\mathsf{includes}(a)$$

evaluates to true in $\overline{M}$.

6. ANALYSIS EXAMPLES

In this section we give a collection of examples that illustrates how one can formalize and analyze different kinds of *authorization questions* about SecureUML+ComponentUML models M using the OCL operations defined in Section 5. The questions are formalized as query operations over objects in $\overline{M}$, possibly with additional arguments also gathered from the objects of $\overline{M}$.

Example: Given a role, what are the atomic actions a user in this role can perform?

```
context Role::allAtomics():Set(Action) body:
    self.allPermissions().allAction()->asSet()
        ->select(a|a.oclIsKindOf(AtomicAction))
```

Example: Given an atomic action, which roles can perform this action?

```
context AtomicAction::allAssignedRoles():Set(Roles) body:
    self.compactionPlus().isassigned.allRoles()->asSet()
```

Example: Given a role and an atomic action, under which circumstances can a user in this role perform this action?

```
context Role::allAuthConst(a:Action):Set(String) body:
  self.permissionPlus(a).isconstraintby.body->asSet()

context Role::permissionPlus(a:Action):Set(Permission) body:
  self.allPermissions()
      ->select(p|p.allActions()->includes(a))
```

Example: Are there two roles with the same set of atomic actions?

```
context Role::duplicateRoles():Boolean body:
    Role.allInstances()
        ->exists(r1, r2| r1.allAtomics = r2.allAtomics)
```

Example: Are there two roles such that one includes the set of actions of the other, but the roles are not related in the role hierarchy?

```
context Role::virtualSubroles():Boolean body:
    Role.allInstances()->exists(r1, r2|
        r1.allActions()->includesAll(r2.allActions())
        and not(r1.superrolePlus()->includes(r2)))
```

Example: Given an atomic action, which roles allows the least set of actions including the atomic action? This requires a suitable definition of "least" and we use here the smallest number of atomic actions.

```
context AtomicAction::minimumRole():Set(Role) body:
    self.allAssignedRoles()->select(r1|self.allAssignedRoles()
      ->forAll(r2| r1.allAtomics()->size()
                   <= r2.allAtomics()->size()))
```

Example: Give the list of "virtual" roles (i.e., roles which have no explicit permissions on their own, but only inherit permissions from other roles).

```
context Role::virtualRoles():Set(Role) body:
    Role.allInstances()->select(r|r.haspermission
      ->isEmpty())
```

Example: Given two atomic actions, are they "distinguishable" (i.e., is there a role that has permission for one, but not the other)?.

```
context AtomicAction::distinguishable(a:AtomicAction):Boolean
body: self.allAssignedRoles() <> a.allAssignedRoles()
```

Example: Given two atomic actions, does the permission for one imply the permission for the other?

```
context AtomicAction::impliedBy(a:AtomicAction):Boolean
body: self.allAssignedPermissions()
      ->forAll(p|p.allActions->includes(a))
```

Example: Do two permissions overlap?

```
context Permission::overlapsWith(p:Permission):Boolean
body: self.allActions()
      ->intersection(p.allActions())->notEmpty()
```

Example: Given a role, are there overlapping permissions?

```
context Role::overlaps():Boolean body:
    self.allPermissions()->exists(p1,p2|
        p1 <> p2 and p1.overlapsWith(p2))
```

Example: Are there overlapping permissions for different roles?

```
context Permission::existOverlapping():Boolean body:
    Permission.allInstances()->exists(p1,p2|
        p1 <> p2 and p1.overlapsWith(p2)
        and not(p1.allRoles->includesAll(p2.allRoles)))
```

Example: Are there duplicate permissions (the same sets of actions, sets of roles, and constraints)?

```
context Permission::duplicate():Boolean body:
    Permission.allInstances()->exists(p1,p2|p1 <> p2 and
        p1.allActions() = p2.allActions() and
        p1.allRoles() = p2.allRoles() and
        p1.isconstraintby = p2.isconstraintby)
```

Example: Are there atomic actions which every role, except the default role, may perform?

```
context AtomicAction::accessAll():Boolean body:
    AtomicAction.allInstances()->exists(a|
        Role.allInstances->forAll(r|
            not(r.default) implies
                r.allAtomics()->includes(a)))
```

The above examples provide evidence that OCL expressions can be used to formalize and check non-trivial security properties. This expressiveness is due to the fact that, in our applications, the OCL language is *enriched* with the types provided by the metamodel of SecureUML+ComponentUML, (e.g, Role, Permission, Set(Action)) and vocabulary (e.g., hasrole, givesaccess, isassigned).

7. THE SECUREMOVA TOOL

As [13] observed, although there are several proposals for specifying role-based authorization constraints, "there is a lack of appropriate tool support for the validation, enforcement, and testing of role-based access control policies. Specifically, tools are needed which can be applied quite easily by a policy designer without too much deeper training." In response to this need, [13] shows how to employ the USE system to validate and test access control policies formulated in UML and OCL. We comment on this work in Section 8.

As part of our work, we have implemented a prototype tool called SecureMOVA, for analyzing SecureUML + ComponentUML model, which is directly based on the mapping from models to metamodel instances defined in Appendix B. SecureMOVA is an extension of the MOVA tool, which is a Java IDE for the ITP/OCL tool, a text-input mode validation and analysis tool for UML diagrams with OCL constraints. The ITP/OCL tool in turn provides commands for building class and object diagrams, validating OCL constraints, and evaluating OCL queries. The ITP/OCL tool is written in Maude [4], a rewriting-based programming language that implements (membership) equational logic. Events on the MOVA's worksheets and toolbars are transformed into ITP/OCL's text-input commands and are interpreted and executed in a Maude process running the ITP/OCL tool.

The SecureMOVA tool is a Java IDE for an extension of the ITP/OCL tool that includes commands for building SecureUML + ComponentUML diagrams and for evaluating OCL queries using, among others, the analysis operations introduced in Sections 5.2 and 6 (the users may, of course, add their own analysis operations to the system). The SecureMOVA tool is still under construction, but the SecureUML + ComponentUML extension of the ITP/OCL tool is already available at `http://maude.sip.ucm.es/securemova`, along with a short tutorial and a collection of examples.

Here we list the ITP/OCL commands used to build the security policy modeled in Figure 4.2:

- (create-security-diagram SCHEDULER) creates the blank security-design diagram SCHEDULER.
- (insert-role SCHEDULER : SupervisorRole) adds the role SupervisorRole to the diagram SCHEDULER; similar commands are used to add the roles SystemAdministratorRole and UserRole.

- (insert-role-hierarchy SCHEDULER | SupervisorRole ↔ UserRole) makes the role UserRole a super-role of the role SupervisorRole in the diagram SCHEDULER.
- (insert-entity SCHEDULER : Meeting) adds the entity Meeting to the diagram SCHEDULER.
- (insert-attribute SCHEDULER : Meeting : Start) adds the attribute Start to the entity Meeting in the diagram SCHEDULER; similar commands are used to add the attribute duration, and the methods Cancel and Notify.
- (insert-permission SCHEDULER : UserMeeting) adds the permission UserMeeting to the diagram SCHEDULER; similar commands are used to add the permissions OwnerMeeting, SupervisorCancel, and ReadMeeting.
- (insert-permission-assignment SCHEDULER : UserMeeting : UserRole) assigns the permission UserMeeting to the role UserRole in the diagram SCHEDULER; similar commands are used to assign the rest of permissions to the appropriate roles.
- (insert-entity-update SCHEDULER : OwnerMeeting : Meeting) assigns update access to all the attributes of the entity Meeting in the diagram SCHEDULER; similar commands are used to assign the different types of access to the entity Meeting to the appropriate permissions.
- (insert-authorization-constraint SCHEDULER : Auth1 : "caller.name=self.owner.name") adds the authorization constraint caller.name=self.owner.name, named Auth1, to the diagram SCHEDULER.
- (insert-authorization-constraint-assignment SCHEDULER : OwnerMeeting : Auth1) assigns the authorization constraint Auth1 tlo the permission OwnerMeeting in the diagram SCHEDULER.

Next, we list different queries that can be formulated to diagram SCHEDULER using the ITP/OCL command query-in. This command takes as an argument the OCL expressions formalizing the query.[2]

- (query-in SEC+COMP-META : SCHEDULER : (defaultRole.existOverlapping)) asks whether there are overlapping permissions for different roles SCHEDULER. ITP/OCL automatically returns the answer true.
- (query-in SEC+COMP-META : SCHEDULER : (MeetingStartAtomicUpdate.allAssignedRoles)) asks which are the roles in the diagram SCHEDULER that are allowed to update the attribute Start of the entity Meeting. ITP/OCL returns the set formed by the roles SupervisorRole and UserRole.
- (query-in SEC+COMP-META : SCHEDULER : (MeetingCancelAtomicExecute.allAssignedPermissions)) asks which are the permissions that allow the execution of the method Cancel of the entity Meeting. ITP/OCL returns the set formed by the permissions SupervisorCancel and OwnerMeeting.

[2]The ITP/OCL tool requires OCL expressions to be written using a notation slightly different from the standard (see `http://maude.sip.ucm.es/securemova`). However, for the sake of the example, we have used here standard OCL notation.

8. CONCLUSION

Related Work. As mentioned in the introduction, our work is inspired by [1], who first explored the use of OCL for querying RBAC policies (see also [14, 13]). A distinct characteristic of our work is that we spell out and follow a precise methodology, which guarantees that query evaluation is formally meaningful. This methodology requires, in particular, precise definitions of both the metamodel of the modeling language and the mapping from models to the corresponding instances of this metamodel. These definitions make it possible to rigorously reason about the meaning of the OCL expressions used in specifying analyzing security policies.

To underscore the importance of such a methodology, consider a simple example: specifying two mutually exclusive roles such as "accounts payable manager" and "purchasing manager". *Mutual exclusion* means that one individual cannot have both roles. In [1, 14, 13] this constraint is specified using OCL as follows:

```
context User inv:
let M : Set = {{accounts payable manager,
                purchasing manager}, ...} in
M->select(m | self.role->
intersection(m)->size > 1)->isEmpty()
```

This constraint correctly specifies mutual exclusion *only if* the association-end role returns all the roles assigned to a user. This should include role assignments explicitly depicted as well as those implicitly assigned to users under the role hierarchy. The actual meaning of the association-end role depends, of course, on the mapping between models and the corresponding instances of the metamodel. Since the precise definition of this mapping is not given in [1, 14, 13], readers (and tool users) must speculate on the meaning of such expressions and thereby the correctness of their OCL specifications. (Notice that, if the mapping used in [1, 14, 13] is the " straightforward" one, the association-end role will only return the roles explicitly assigned to a user.)

In our setting, mutual exclusion can be specified using OCL as follows:

```
context User inv:
let M : Set = {{accounts payable manager,
                purchasing manager}, ...}
in M->select(m | self.hasrole.superrolePlus()
->intersection(m)->size > 1)->isEmpty()
```

From our definition of superrolePlus in Section 5.2, it is clear that this expression denotes all the roles assigned to a user, including those implicitly assigned to the user under the specified role hierarchy.

Future Work. One direction for future work is tool support for handling queries involving system state. Recall that SecureUML includes the possibility of constraining permissions with authorization constraints (OCL formulas), which restrict the permissions to those system states satisfying the constraints. An example of a stateful query for a design metamodel that includes access to the system date is "which operations are possible on week days that are impossible on weekends?" Alternatively, in a banking model, we might ask "which actions are possible on overdrawn bank accounts?" Such queries cannot currently be evaluated as

they require reasoning about consequences of OCL formulas and this involves theorem proving as opposed to model checking, i.e., determining the satisfiability of formulas in a concrete model.

Another interesting direction would be to use our approach to analyze the consistency of different system views. In [2] we showed how one can combine SecureUML with different modeling languages (i.e., ComponentUML and ControllerUML) to formalize different views of multi-tier architectures. In this setting, access control might be implemented at both the middle tier (implementing a controller for, say, a web-based application) and a back-end persistence tier. If the policies for both of these tiers are formally modeled, we can potentially answer question like "will the controller ever enter a state in which the persistence tier throws a security exception?" Again, carrying out such analysis would require support for theorem proving.

9. REFERENCES

[1] Gail-Joon Ahn and Michael E. Shin. Role-based authorization constraints specification using object constraint language. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 157–162, Washington, DC, USA, 2001. IEEE Computer Society.

[2] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[3] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[5] Manuel Clavel, Marina Egea, Rafael Martínez, and Viviane Torres da Silva. The MOVA Tool, 2006. `http://maude.sip.ucm.es/~mova`.

[6] David F. Ferraiolo, Ravi S. Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.

[7] Geri Georg, Indrakshi Ray, and Robert France. Using aspects to design a secure system. In *ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[8] Jan Jürjens. Towards development of secure systems using UMLsec. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE/ETAPS 2001)*, number 2029 in LNCS, pages 187–200. Springer-Verlag, 2001.

[9] Jan Jürjens. UMLsec: Extending UML for secure systems development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 — The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer-Verlag, 2002.

[10] Anneke Kleppe, Wim Bast, Jos B. Warmer, and Andrew Watson. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.

[11] Object Management Group. Object Constraint Language specification, 2004. `http://www.omg.org`.

[12] Object Management Group. Unified Modeling Language specification, 2004. `http://www.uml.org`.

[13] Karsten Sohr, Gail-Joon Ahn, Martin Gogolla, and Lars Migge. Specification and validation of authorisation constraints using UML and OCL. In *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 64–79. Springer-Verlag, 2005.

[14] Hua Wang, Yanchun Zhang, Jinli Cao, and Jian Yang. Specifying role-based access constraints with object constraint language. In *Advanced Web Technologies and Applications*, volume 3007 of *Lecture Notes in Computer Science*, pages 687–696. Springer Berlin-Heidelberg, 2004.

[15] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2st edition, 2003.

APPENDIX

A. THE SECUREUML + COMPONENTUML METAMODEL CONSTRAINTS

Default role. The following invariants guarantee that models contain a *default* role with the desired semantics; in particular, any user is assigned, at least, the *default* role.

```
context Role
inv existsADefaultRole:
    self.allInstances()->select(r|r.default)->size() = 1
inv allRolesInheritFromDefaultRole:
    self.superrolePlus()->exists(r|r.default)

context User
inv allUsersAssignedDefaultRole:
    self.hasrole->exists(r|r.default)
```

Role hierarchy. The following invariant guarantee that there are no cycles in the role hierarchy.

```
context Role inv noCyclesinRoleHierarchy:
  self.superrole->forAll(r|r.superrolePlus()->excludes(self))
```

Default permission. The following invariants guarantee that models contain a *default* permission with the desired semantics. In particular, the default permission may only be given to the default role, and may contain only atomic actions. Also, atomic actions are assigned at least one permission and, if they are assigned more than one, then none of them can be the default permission.

```
context Permission
inv existsADefaultPermission:
    self.allInstances()->select(p|p.default)->size() = 1
```

```
inv defaultPermissionAssignedToDefaultRole:
    self.default implies self.givesaccess->forAll(r|r.default)

context CompositeAction inv nonDefaultPermission:
    self.allAssignedPermission->forAll(p|not(p.default))

context AtomicAction
inv existsAPermission:
    self.allAssignedPermission()->notEmpty()
inv overridingDefaultPermission:
    self.allAssignedPermission()
    ->forAll(p1, p2| p1<>p2 implies not(p1.default))
```

Resource action association. The following invariants guarantees that actions refer to the correct resource.

```
context AtomicCreate inv targetsAnEntity:
  self.resource.oclIsTypeOf(Entity)
context AtomicDelete inv targetsAnEntity:
  self.resource.oclIsTypeOf(Entity)
context AtomicUpdate inv targets:
  self.resource.oclIsTypeOf(Attribute)
  or self.resource.oclIsTypeOf(AssociationEnd)
context AtomicRead inv targets:
  self.resource.oclIsTypeOf(Attribute)
  or self.resource.oclIsTypeOf(AssociationEnd)
context AtomicExecute inv targetsAMethod:
  self.resource.oclIsTypeOf(Method)
context EntityFullAccess inv targetsAnEntity:
  self.resource.oclIsTypeOf(Entity)
context EntityRead inv targetsAnEntity:
  self.resource.oclIsTypeOf(Entity)
context EntityUpdate inv targetsAnEntity:
  self.resource.oclIsTypeOf(Entity)
context AttributeFullAccess inv targetsAnAttribute:
  self.resource.oclIsTypeOf(Attribute)
context AssociationEndFullAccess
inv targetsAnAssociationEnd:
  self.resource.oclIsTypeOf(AssociationEnd)
```

The following constraints ensure that resources have the correct actions defined on them.

```
context Entity inv areAccessedBy:
  self.action->size() = 5 and
  self.action->exists(a|a.oclIsTypeOf(EntityFullAccess)) and
  self.action->exists(a|a.oclIsTypeOf(EntityUpdate)) and
  self.action->exists(a|a.oclIsTypeOf(EntityRead)) and
  self.action->exists(a|a.oclIsTypeOf(AtomicCreate)) and
  self.action->exists(a|a.oclIsTypeOf(AtomicDelete))

context Attribute inv areAccessedBy:
  self.action->size() = 3 and
  self.action->exists(a|
      a.oclIsTypeOf(AttributeFullAccess)) and
  self.action->exists(a|a.oclIsTypeOf(AtomicRead)) and
  self.action->exists(a|a.oclIsTypeOf(AtomicUpdate))

context Method inv areAccessedBy:
  self.action->size() = 1 and
  self.action->exists(a|a.oclIsTypeOf(AtomicExecute))

context Association-end inv areAccessedBy:
  self.action->size() = 3 and
  self.action->exists(a|
      a.oclIsTypeOf(AssociationEndFullAccess)) and
  self.action->exists(a|a.oclIsTypeOf(AtomicRead)) and
  self.action->exists(a|a.oclIsTypeOf(AtomicUpdate))
```

Action Hierarchy. The following invariants guarantee that composite actions are composed of the correct subordinated actions.

```
context EntityFullAccess inv containsSubactions:
    self.subordinatedactions = self.resource.action
    ->select(a|a.oclIsTypeOf(EntityUpdate))
    ->union(self.resource.action
            ->select(a|a.oclIsTypeOf(EntityRead)))
    ->union(self.resource.action
            ->select(a|a.oclIsTypeOf(AtomicCreate)))
    ->union(self.resource.action
            ->select(a|a.oclIsTypeOf(AtomicDelete)))

context EntityRead inv containsSubactions:
    self.subordinatedactions =
    self.resource.oclAsType(Entity).hasattribute.action
    ->select(a|a.oclIsTypeOf(AtomicRead))
    ->union(self.resource.oclAsType(Entity)
            .hasassociationend.action
            ->select(a|a.oclIsTypeOf(AtomicRead)))
    ->union(self.resource.oclAsType(Entity).hasmethod
            ->select(me|me.isQuery).action
            ->select(a|a.oclIsTypeOf(AtomicExecute)))

context EntityUpdate inv containsSubactions:
    self.subordinatedactions =
    self.resource.oclAsType(Entity).hasattribute.action
    ->select(a|a.oclIsTypeOf(AtomicUpdate))
    ->union(self.resource.oclAsType(Entity)
            .hasassociationend.action
            ->select(a|a.oclIsTypeOf(AtomicUpdate)))
    ->union(self.resource.oclAsType(Entity).hasmethod
            ->select(me|not(me.isQuery)).action
            ->select(a|a.oclIsTypeOf(AtomicExecute)))

context AttributeFullAccess inv containsSubactions:
    self.subordinatedactions = self.resource.action
    ->select(a|a.oclIsTypeOf(AtomicUpdate))
    ->union(self.resource.action
            ->select(a|a.oclIsTypeOf(AtomicRead)))

context AssociationEndFullAccess inv containsSubactions:
    self.subordinatedactions = self.resource.action
    ->select(a|a.oclIsTypeOf(AtomicUpdate))
    ->union(self.resource.action
            ->select(a|a.oclIsTypeOf(AtomicRead)))
```

B. MAPPING TABLES

We here give an informal, but nevertheless complete, definition of the mapping from SecureUML + componentUML concrete syntax to the corresponding instance of the abstract syntax metamodel.

Insert an object "default" of the class Role, with value true for its default-attribute.
For each role r, insert (i) an object $\overline{r}$ of the class Role, with value false for its default-attribute, and (ii) a RoleHierarchy-link between $\overline{r}$ (subrole) and "default".
For each inheritance relationship between two roles r_1 (subrole) and r_2, insert a RoleHierarchy-link between $\overline{r_1}$ (subrole) and $\overline{r_2}$.
For each user u, insert (i) an object $\overline{u}$ of the class User, and (ii) a UserAssignment-link between $\overline{u}$ and the object "default" of the class Role.
For each assignment of a user u to a role r, insert a UserAssignment-link between $\overline{u}$ and $\overline{r}$.
For each entity e, insert (i) an object $\overline{e}$ of the class Entity; (ii) an object $efa(\overline{e})$ of the class EntityFullAccess; (iii) an object $eu(\overline{e})$ of the class EntityUpdate; (iii) an object $er(\overline{e})$ of the class EntityRead; (iii) an object $ac(\overline{e})$ of the class AtomicCreate; (iv) an object $ad(\overline{e})$ of the class AtomicDelete; (v) ResourceAssignment-links between $\overline{e}$ and $efa(\overline{e})$, $\overline{e}$ and $eu(\overline{e})$, $\overline{e}$ and $er(\overline{e})$, $\overline{e}$ and $ac(\overline{e})$, and $\overline{e}$ and $ad(\overline{e})$; (vi) ActionHierarchy-links between $eu(\overline{e})$ (subordinatedAction) and $efa(\overline{e})$, $er(\overline{e})$ (subordinatedAction) and $efa(\overline{e})$, $ac(\overline{e})$ (subordinatedAction) and $efa(\overline{e})$, and $ad(\overline{e})$ (subordinatedAction) and $efa(\overline{e})$; and (vii) ActionAssignment-links between $ac(\overline{e})$ and $ad(\overline{e})$ and the object "default" of the class Permission.
For each attribute a of an entity e, insert (i) an object $\overline{a}$ of the class Attribute; (ii) an object $afa(\overline{a})$ of the class AttributeFullAccess; (iii) an object $au(\overline{a})$ of the class AtomicUpdate; (iii) an object $ar(\overline{a})$ of the class AtomicRead; (vi) ResourceAssignment-links between $\overline{a}$ and $afa(\overline{a})$, $\overline{a}$ and $au(\overline{a})$, and $\overline{a}$ and $ar(\overline{a})$; (iv) ActionHierarchy-links between $au(\overline{a})$ (subordinatedAction) and $afa(\overline{e})$, $ar(\overline{a})$ (subordinatedAction) and $afa(\overline{e})$, $au(\overline{a})$ (subordinatedAction) and $eu(\overline{e})$, and $ar(\overline{a})$ (subordinatedAction) and $er(\overline{e})$; (v) an EntityAttibute-link between $\overline{e}$ and $\overline{a}$; and (vi) ActionAssignment-links between $au(\overline{a})$ and $ar(\overline{a})$ and the object "default" of the class Permission.
For each query method m of an entity e, insert (i) an object $\overline{m}$ of the class Method, with value true for its isQuery-attribute; (ii) an object $ae(\overline{m})$ of the class AtomicExecute; (iii)a ResourceAssignment-link between $\overline{m}$ and $ae(\overline{m})$; (iv) anActionHierarchy-link between $ae(\overline{m})$ (subordinatedAction) and $er(\overline{e})$; (v) an EntityMethod-link between $\overline{e}$ and $\overline{m}$; and (vi) an ActionAssignment-link between $ae(\overline{m})$ and the object "default" of the class Permission.
For each non-query method m of an entity e, insert (i) an object $\overline{m}$ of the class Method, with value false for its isQuery-attribute; (ii) an object $ae(\overline{m})$ of the class AtomicExecute; (iii) a ResourceAssignment-link between $\overline{m}$ and $ae(\overline{m})$; (iv) an ActionHierarchy-link between $ae(\overline{m})$ (subordinatedAction) and $eu(\overline{e})$; (v) an EntityMethod-link between $\overline{e}$ and $\overline{m}$; and (vi) an ActionAssignment-link between $ae(\overline{m})$ and the object "default" of the class Permission.
For each association-end d of an entity e, insert (i) an object $\overline{d}$ of the class AssociationEnd; (ii) an object $dfa(\overline{d})$ of the class AssociationEndFullAccess; (iii) an object $au(\overline{d})$ of the class AtomicUpdate, (iii) an object $ar(\overline{d})$ of the class AtomicRead; (v) ResourceAssignment-links between $\overline{d}$ and $dfa(\overline{d})$, $\overline{d}$ and $au(\overline{d})$, and $\overline{d}$ and $ar(\overline{d})$; (iv) ActionHierarchy-links between $au(\overline{d})$ (subordinatedAction) and $dfa(\overline{d})$, $ar(\overline{a})$ (subordinatedAction) and $dfa(\overline{d})$, $au(\overline{d})$ (subordinatedAction) and $eu(\overline{e})$, and $ar(\overline{d})$ (subordinatedAction) and $er(\overline{e})$; (v) an EntityAssociationEnd-link between $\overline{e}$ and $\overline{d}$; and (vi) ActionAssignment-links between $au(\overline{d})$ and $ar(\overline{d})$ and the object "default" of the class Permission.

Table 1: Mapping from models to metamodel instances (I).

Insert an object "default" of the class AuthorizationConstraint, with values "OCL" and "true", respectively, for its language and **body** attributes.
For each authorization constraint ath, insert an object $\overline{ath}$, with values "OCL" and "ath" for its language and **body** attributes, respectively.
Insert an object "default" of the class Permission, with value true for its default-attribute.
Insert a ConstraintAssignment-link between the "default" object of the class Permission and the "default" object of the class AuthorizationConstraint.
Insert a PermissionAssignment-link between the "default" object of the class Permission and the "default" object of the class Role.
For each permission p, insert an object $\overline{p}$ of the class Permission, with value false for its default-attribute.
For each assignment of a permission p to a role r, insert a PermissionAssignment-link between $\overline{r}$ and $\overline{p}$.
For each assignment of a constraint ath to a permission p, insert a ConstraintAssignment-linke between $\overline{p}$ and $\overline{ath}$.
For each entity e, and each permission p that grants "entity full access" to e, insert an ActionAssignment-link between $efa(\overline{e})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $ac(\overline{e})$ or $ad(\overline{e})$. For each attribute a, method m, and association-end d of e, delete any ActionAssignment-link between the "default" object of the class Permission and $au(\overline{a})$, $ar(\overline{a})$, $au(\overline{d})$, $ar(\overline{d})$, or $ae(\overline{m})$.
For each entity e, and each permission p that grants "entity read access" to e, insert an ActionAssignment-link between $er(\overline{e})$ and $\overline{p}$. For each attribute a, query method m, and association-end d of e, delete any ActionAssignment-link between the "default" object of the class Permission and $ar(\overline{a})$, $ar(\overline{d})$, or $ae(\overline{m})$.
For each entity e, and each permission p that grants "entity update access" to e, insert an ActionAssignment-link between $eu(\overline{e})$ and $\overline{p}$. For each attribute a, non-query method m, and association-end d of e, delete any ActionAssignment-link between the "default" object of the class Permission and $au(\overline{a})$, $au(\overline{d})$, or $ae(\overline{m})$.
For each entity e, and each permission p that grants "atomic create access" to e, insert an ActionAssignment-link between $ac(\overline{e})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $ac(\overline{e})$.
For each entity e, and each permission p that grants "atomic delete access" to e, insert an ActionAssignment-link between $ad(\overline{e})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $ad(\overline{e})$.
For each attribute a, and each permission p that grants "attribute full access" to a, insert an ActionAssignment-link between $afa(\overline{a})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $au(\overline{a})$ or $ar(\overline{a})$.
For each attribute a, and each permission p that grants "atomic update access" to a, insert an ActionAssignment-link between $au(\overline{a})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $au(\overline{a})$.
For each attribute a, and each permission p that grants "atomic read access" to a, insert an ActionAssignment-link between $ar(\overline{a})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $ar(\overline{a})$.
For each association-end d, and each permission p that grants "association-end full access" to d, insert an ActionAssignment-link between $dfa(\overline{d})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $au(\overline{d})$ or $ar(\overline{d})$.
For each association-end d, and each permission p that grants "atomic update access" to d, insert an ActionAssignment-link between $au(\overline{d})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $au(\overline{d})$.
For each association-end d, and each permission p that grants "atomic read access" to d, insert and ActionAssignment-link between $ar(\overline{d})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $ar(\overline{d})$.
For each method m, and each permission p that grants "atomic execute access" to m, insert an ActionAssignment-link between $ae(\overline{m})$ and $\overline{p}$. Delete any ActionAssignment-link between the "default" object of the class Permission and $ae(\overline{m})$.

Table 2: Mapping from models to metamodel instances (II).