A Metamodel-Based Approach for Analyzing Security-Design Models

David Basin[1], Manuel Clavel[2], Jürgen Doser[1], and Marina Egea[2]

[1] Information Security Group, ETH Zurich
{basin,doserj}@inf.ethz.ch
[2] Computer Science Department, U. Complutense, Madrid
clavel@sip.ucm.es, marina_egea@fdi.ucm.es

Abstract We have previously proposed an expressive UML-based language for constructing and transforming security-design models, which are models that combine design specifications for distributed systems with specifications of their security policies. 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 non-trivial security properties of security-design models. The approach and examples presented have been implemented and checked in the SecureMOVA tool.

1 Introduction

Model driven development [9] holds the promise of reducing system development time and improving the quality of the resulting products. Recent investigations [2,6,7,8] have shown that security can be integrated into system-design models and that the resulting *security-design models* can be used to generate systems along with their security infrastructures. Moreover, when the models have a formal semantics, they can be reasoned about: one can query their properties and understand potentially subtle consequences of the policies they define.

In previous work [2], we presented a UML-based security modeling language, called SecureUML, closely related to Role Based Access Control (RBAC) [5]. We showed how to systematically combine different design modeling languages with SecureUML in a way that allows users to formalize authorization restrictions on systems implementing the design. The combination scheme was defined both syntactically and semantically and we also 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. In our setting, security-design models constitute formal objects with both a concrete syntax (or notation) and an abstract syntax. Security-design models themselves are described by a metamodel that

formalizes the structure of well-formed models. We show that, in this setting, security properties of security-design models can be expressed as formulas in the Object Constraint Language (OCL) [11] over this metamodel. We can formalize queries in this language that ask questions about the relationships between users, roles, permissions, and actions. An example of a typical query (taken from Section 5) 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 the OCL expressions over the metamodel of the security modeling language.

The idea of formulating OCL queries about role-based access control policies is not new. Our work is inspired by [1,12], who first explored the use of OCL for querying RBAC policies, and we make comparisons in Section 7. 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 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 SecureMOVA, a security-design modeling tool whose implementation is directly based on our metamodel-based approach for analyzing security-design models.

2 General Approach

Background: models and meaning. A modeling language provides a vocabulary (concepts and relations) for building models, as well as a notation to graphically depict them as diagrams. Diagrams have to conform with the metamodel of the modeling language. 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 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 the evaluation of OCL expressions. If the mapping is not explicitly given or the requirements are not satisfied, then the validity of the results may be open, or even wrong (for examples, see the related work section).

To be more precise, we state the following requirements. 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:

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

In this diagram, the downward arrow on the left side denotes the evaluation of the OCL expression exp_f (the result of which, denoted by the function $ev(\_,\_)$, constitutes another abstract model). The downward 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 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, its invariants, and of the underlying mapping from models to the corresponding instances of the metamodel. Note that this is already a necessary condition for stating meaningful OCL expressions on models in the first place.

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 [14], "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, 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.

3 SecureUML+ComponentUML

In this section, we describe SecureUML and Component UML, the security and design modeling languages that we use to illustrate our approach and some of its key aspects, like the mapping from models to instances of the metamodel.

3.1 The SecureUML+ComponentUML Metamodel

SecureUML [2,10] is a modeling language for formalizing access control requirements that is based on RBAC [5]. In RBAC, permissions specify which *roles* are authorized to perform given operations. RBAC additionally allows one to organize these roles in a hierarchy, where roles can inherit permissions along the hierarchy. In this way, the security policy can be described closely following the hierarchical structure of an organization. Users are then granted permissions by being assigned to the appropriate roles, based on their competencies and responsibilities in the organization.

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 are specified in a so called *dialect* and depend on the primitives for constructing models in the system-design modeling language. Figure 1 shows the SecureUML metamodel. The system-design modeling language that we consider here, ComponentUML, 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 metamodel of ComponentUML is shown in the right part of Figure 2. The dialect definition, shown in the left part of Figure 2, additionally specifies:

- The model element types of the system-design modeling language that represent protected resources. Here, *Entities*, as well as their *Attributes*, *Methods*, and *AssociationEnds* (but not *Associations* as such) are protected resources.

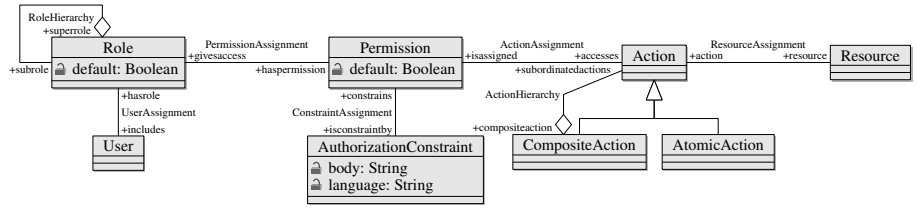

Figure 1. SecureUML Metamodel.

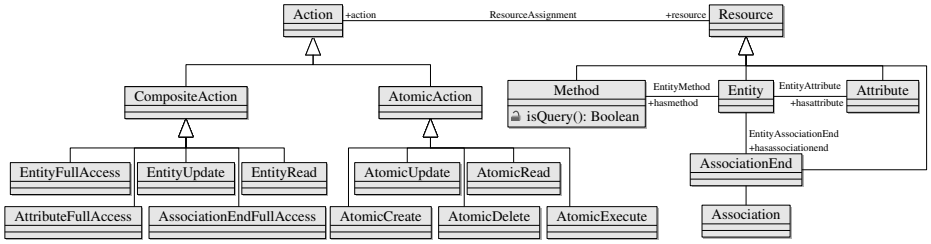

Figure 2. ComponentUML Dialect Metamodel.

– The actions these resources types offer and hierarchies classifying these actions. The actions offered here are shown in the following table:

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

The atomic actions are intended to directly map onto actual operations of the modeled system. The composite actions are used to group more primitive actions into a hierarchy of more higher-level ones. Here, for example, the composite action *AttributeFullAccess* contains both the read and the update action of the attribute. The precise definition of the actions offered by the different resources, and their hierarchical relationship, is made by adding OCL invariants to the metamodel. The interested reader can find the complete list of these constraints in the references given at `http://maude.sip.ucm.es/securemova`.

– the default access control policy for actions where no explicit permissions is defined (i.e., whether access is allowed or denied by default). Here, by default, access is allowed.

3.2 The SecureUML+ComponentUML Models

We use Figure 3 as a running example to illustrate the concrete syntax of SecureUML and ComponentUML. In this example, the system should maintain a

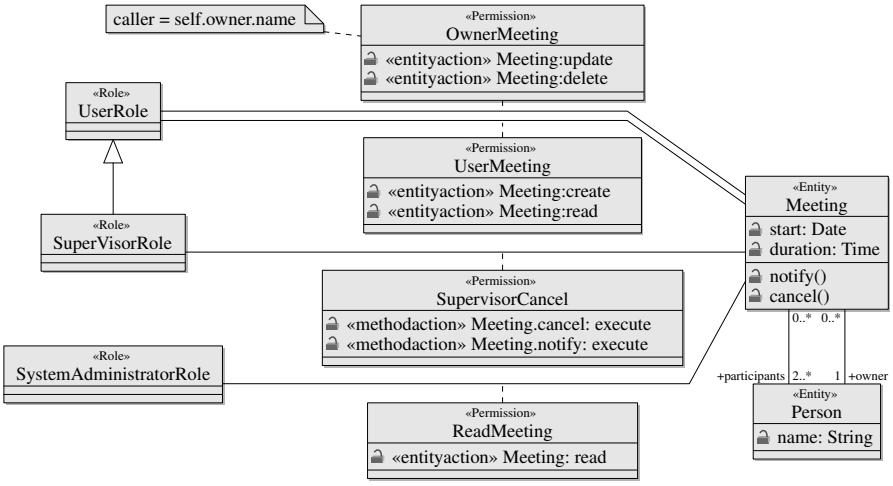

Figure 3. Example Security Policy.

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 3 formalizes this security policy using the UML profile for SecureUML and ComponentUML defined in [2]. In this profile, 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. The stereotype ≪entityaction≫, for example, specifies

that a permission attribute refers to an action on an entity. 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. The authorization constraint expressions are attached to the permissions' association classes. ComponentUML 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.

3.3 The Mapping From Models to Metamodel Instances

Recall that, in our approach, the specification of security properties using OCL directly depends on the mapping from models to instances of the metamodel, since the expressions formalizing the properties will not be evaluated over the graphical 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 associations between UML model elements are mapped to appropriate links between the instances of the corresponding metamodel elements.

In some cases, however, this mapping is less straightforward, in particular, where the notation provides the modeler with convenient "syntactic sugar". 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, however, 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, although they may not be mentioned in the attributes of the permissions depicted in M.
- Links between the "default" objects of type Role, AuthorizationConstraint, and Permission, and between the "default" object of type Permission and the objects of subtypes of Action, which correspond to the default access control policy defined in the metamodel.
- Links between the objects of subtypes of Action, which correspond to the hierarchy of actions defined in the metamodel.

The reader can find the complete definition of this mapping in the references given at `http://maude.sip.ucm.es/securemova`.

4 Analyzing SecureUML+ComponentUML Models

In this section, we define OCL query operations over the metamodel of SecureUML+ComponentUML that capture different aspects of the access control

information contained in the models. These operations will be 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.[3] 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.

4.1 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 $\Sigma_{RBAC} = (\mathcal{S}_{RBAC}, \geq_{RBAC}, \mathcal{F}_{RBAC}, \mathcal{P}_{RBAC})$ 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, $\geq_{RBAC}$ 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 $Actions \geq_{RBAC} AtomicActions$. Also, let $\mathcal{F}_{RBAC} = \emptyset$ and

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

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. The relation $\geq_{Roles}$ is given by the reflexive and transitive closure of the association RoleHierarchy on Role, and the relation $\geq_{Actions}$ by the reflexive and transitive closure of the association ActionHierarchy. Finally, the relations *UA*, *PA*, and *AA* contain tuples for each instance of the associations UserAsssignment, PermissionAssignment, and ActionAssignment.

Note that the SecureUML metamodel and its semantics mention "users" and user assignments to roles. These are not usually modeled in the security-design model (e.g., they are not depicted in Figure 3) because this is configuration data that is typically not known at modeling time. For analysis or illustrative purposes, such configuration data can be given as additional input.

Remark 1. Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any u in *Users*, p in *Permissions*, and a in *Actions*, the following table shows the basic correspondence between satisfaction in $\Im_{RBAC}$ and evaluation of OCL expressions in $\overline{M}$:

[3] 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 OCL evaluators.

is satisfied in $\Im_{RBAC}$	evaluates to true over $\overline{M}$
$UA(u, r)$	u.hasrole−>includes(r)
$PA(r, p)$	r.haspermission−>includes(p)
$AA(p, a)$	p.accesses−>includes(a)

4.2 Analysis Operations

In this section, we define a number of OCL *query operations* that are useful for analyzing security properties of security-design models formalized using SecureUML+ComponentUML. We also mention other OCL analysis operations, whose definitions we omit here.

To analyze the relation $\geq_{Roles}$, we define Role::superrolePlus():Set(Role), which is an operation that 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.superrole))
  else rs−>including(self)
  endif
```

Similarly, we define the operation Role::subrolePlus():Set(Role) returning the roles (directly or indirectly) *below* a given role in the role hierarchy. Also, we use these operations to define the operation Role::allPermissions():Set(Permission) that returns the collection of permissions (directly or indirectly) assigned to a role.

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

Conversely, we define the operation Permission::allRoles():Set(Role), returning the collection of roles (directly or indirectly) assigned to the given permission.

To analyze the relation $\geq_{Actions}$, we define Action::subactionPlus():Set(Action) that 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
```

Similarly, we define the operation Action::compactionPlus():Set(Action) returning the collection of actions to which an action is (directly or indirectly) subordinated. In addition, we define the operation Permission::allActions():Set(Action) that

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()
```

Conversely, we define the operation Action::allAssignedPermissions():Set(Permission), returning the collection of permissions that (directly or indirectly) grant access to an action. Finally, we define the operation User::allAllowedActions():Set(Action) that returns the collection of actions that are permitted for the given 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 2. Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M. Then, for any u in *Users*, r, r_1, r_2 in *Roles*, p in *Permissions*, and a, a_1, a_2 in *Actions*, the following table shows the additional correspondence between satisfaction in $\Im_{RBAC}$ and evaluation of OCL expressions in $\overline{M}$.

is satisfied in $\Im_{RBAC}$	evaluates to true in $\overline{M}$
$r_1 \geq_{Roles} r_2$	r_2.superrolePlus()->includes(r_1) r_1.subrolePlus()->includes(r_2)
$\exists r_2 \in Roles.\ r_2 \geq_{Roles} r_1 \wedge PA(r_2, p)$	r_1.allPermissions()->includes(p) p.allRoles()->includes(r_1)
$a_1 \geq_{Actions} a_2$	a_1.subactionPlus()->includes(a_2) a_2.compactionPlus()->includes(a_1)
$\exists a_2 \in Actions.\ a_2 \geq_{Actions} a_1 \wedge AA(p, a_2)$	p.allActions->includes(a_1) a_1.allAssignedPermisssions()->includes(p)
$\phi_{RBAC}(u, a)$	u.allAllowedActions()->includes(a)

Here, $\phi_{RBAC}(u, a)$ is the formula that states whether a user u has a permission to perform action a:

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

5 Analysis Examples

In this section, we give a collection of examples that illustrates how one can analyze SecureUML+ComponentUML models M using the OCL operations defined in Section 4. The questions are formalized as queries over objects in $\overline{M}$, possibly with additional arguments. Note that, with the exception of Example

3, the queries refer to static information about the access control configuration, which is independent of the system state. In contrast, in Example 3 we explicitly query about the circumstances under which a user can perform an action.

The first three examples address the basic question of who can do what, under which circumstances. These functions can provide an elementary sanity check of the access control policy.

Example 1. Given a role, which atomic actions can a user in this role perform?

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

Example 2. Given an atomic action, which roles can perform this action?

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

Example 3. 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))
```

The next two examples address the question of whether there are possibilities for refactoring or simplifying the role hierarchy. If we have two roles with the same set of allowed actions, one of them may be redundant and could therefore be removed. Similarly, consider two roles where one role is allowed everything the other role is allowed. In this case, the policy could be simplified by letting the second role inherit from the first.

Example 4. 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 5. 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)))
```

The next example addresses the question of which role a user should be assigned, given that he is supposed to perform a particular action. According to the least-privilege-principle, the user should have no more privileges than absolutely required.

Example 6. Given an atomic action, which roles allow 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()))
```

The next two examples address the question of whether there are possibilities for refactoring permissions. Given two permissions that share allowed actions, it may be useful to refactor the common actions into a new, separate permission.

Example 7. Do two permissions overlap?

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

Example 8. 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)))
```

The next example provides another way of detecting opportunities for refactoring permissions. Suppose the policy default is to allow access and, moreover, there is an action that is allowed by every role. The policy can then be simplified by removing this action from all permissions, effectively assigning it the default permission.

Example 9. Are there atomic actions that 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).

6 The SecureMOVA Tool

As [12] observed, although there have been various proposals for specifying role-based authorization constraints, there is a lack of appropriate tool support for analyzing role-based access control policies. In response to this need, [12] 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 7.

As part of our work, we have implemented a prototype tool called SecureMOVA for analyzing SecureUML+ComponentUML models. SecureMOVA is an extension of the ITP/OCL tool, a text-input mode validation and analysis tool for UML diagrams with OCL constraints. SecureMOVA extends the ITP/OCL tool with commands for building SecureUML+ComponentUML diagrams and for evaluating OCL queries using, among others, the analysis operations introduced in Sections 4.2 and 5 (the users may, of course, add their own analysis operations to the system). Importantly, SecureMOVA implements the mapping from models to instances of the metamodel introduced in Section 3.3. Thus, the users can work directly with the models (as they are used to), but their queries are evaluated over the corresponding instances of the metamodel, which are automatically generated by SecureMOVA. For reason of space, we omit here the complete definition of the SecureMOVA commands. The interested reader can find it at `http://maude.sip.ucm.es/securemova` along with a collection of examples, including the example in Figure 3.

7 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 [13,12]). 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 and 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 [1,13,12] 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 [1,13,12], 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 [1,13,12] 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 4.2, 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.

OCL has also been used to analyze models of other modeling languages, not only security modeling languages. For example, consider the use of OCL to define metrics, originally proposed by [4]. These approaches share the problems we elaborated in Section 2: without a precise relation between the graphical models and the corresponding metamodel, and a precise relation to the semantic domain, the meaning and validity of OCL formulas is unclear.

Future Work One direction for future work is tool support for handling queries involving system state. 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.

References

1. G. J. Ahn and M. 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. D. A. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, January 2006.
3. R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. 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. F. Brito e Abreu. Using OCL to formalize object oriented metrics definitions. Technical Report ES007/2001, FCT/UNL and INESC, Portugal, June 2001. available at `http://ctp.di.fct.unl.pt/QUASAR/Resources/Papers/others/MOOD_OCL.pdf`.
5. D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for Role-Based Access Control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, August 2001.
6. G. Georg, I. Ray, and R. France. Using aspects to design a secure system. In *ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, pages 117–126, Washington, DC, USA, 2002. IEEE Computer Society.
7. J. Jürjens. Towards development of secure systems using UMLsec. In *Fundamental Approaches to Software Engineering (FASE/ETAPS 2001)*, volume 2029 of *LNCS*, pages 187–200. Springer, 2001.
8. J. Jürjens. UMLsec: Extending UML for secure systems development. In *UML 2002 — The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer, 2002.
9. A. Kleppe, W. Bast, J. B. Warmer, and A. Watson. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.
10. Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML: A UML-based modeling language for model-driven security. In *UML 2002 — The Unified Modeling Language*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.
11. Object Management Group. Object Constraint Language specification, version 2.0, May 2006.
12. K. Sohr, G. J. Ahn, M. Gogolla, and L. Migge. Specification and validation of authorisation constraints using UML and OCL. In *Computer Security – ESORICS 2005*, volume 3679 of *LNCS*, pages 64–79. Springer, 2005.
13. H. Wang, Y. Zhang, J. Cao, and J. Yang. Specifying Role-Based Access Constraints with Object Constraint Language. In *Advanced Web Technologies and Applications*, volume 3007 of *LNCS*, pages 687–696. Springer, 2004.
14. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2nd edition, 2003.