Automated Analysis of Security-Design Models

David Basin [a] Manuel Clavel [b,1] Jürgen Doser [a] Marina Egea [b,1]
[a] *Information Security Group, ETH Zurich*
[b] *Computer Science Department, Universidad Complutense Madrid*

Abstract

We have previously proposed SecureUML, an expressive UML-based language for constructing security-design models, which are models that combine design specifications for distributed systems with specifications of their security policies. Here we show how to automate the analysis of such models in a semantically precise and meaningful way. In our approach, models are formalized together with scenarios that represent possible run-time instances. Queries about properties of the security policy modeled are expressed as formulas in UML's Object Constraint Language. The policy may include both declarative aspects, i.e., static access-control information such as the assignment of users and permissions to roles, and programmatic aspects, which depend on dynamic information, namely the satisfaction of authorization constraints in the given scenario. We show how such properties can be evaluated, completely automatically, in the context of the metamodel of the security-design language. We demonstrate, through examples, that this approach can be used to formalize and check non-trivial security properties. The approach has been implemented in the SecureMOVA tool and all of the examples presented have been checked using this tool.

Key words: UML, OCL, SecureUML, Access Control Policies, Security Policies, Formal Analysis, Metamodels

1. Introduction

Model driven development [11] holds the promise of reducing system development time and improving the quality of the resulting products. Recent investigations [4,8,9,10] have shown that security policies 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 previous work [4], we presented a security modeling language, called SecureUML, closely related to Role Based Access Control (RBAC). We showed how to combine this language (both syntactically and semantically) with different design languages, thereby creating languages for formalizing *security-design models* that specify system designs together with their security requirements. Our focus in [4] was on the language definitions and generating access-control infrastructure from security-design models. Although we gave SecureUML a formal semantics, we did not investigate methods for formalizing and mechanically analyzing the security properties of security-design models. This is our focus in the present paper.

Security-design models are formal objects with both a concrete syntax (or notation) and an abstract syntax. The modeling language for security-design models is described by a metamodel that formalizes the structure of well-formed models as well as scenarios, which represent possible run-time instances, i.e., concrete system states. We show that, in this setting, security properties of security-design models can be expressed as formulas in OCL [12], the Object Constraint Language of UML, in the context of the metamodel combining SecureUML with a design language. The result is an expressive language for formalizing queries, which utilizes the entire vocabulary present in this metamodel. We can formalize queries about the relationships between users, roles, permissions, actions, and even system states. An example of a typical query about a security policy 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?" Another example, but this time involving system state, is "which roles can be

Email addresses: basin@inf.ethz.ch (David Basin), clavel@sip.ucm.es (Manuel Clavel), doserj@inf.ethz.ch (Jürgen Doser), marina_egea@fdi.ucm.es (Marina Egea).

[1] Research partially supported by Spanish MEC Projects TIN2005-09207-C03-03 and TIN2006-15660-C02-01, and by Comunidad de Madrid Program S-0505/TIC/0407.

assigned to a user so as to allow her to perform an action on a concrete resource in the context of a given scenario?" As we will see, we answer such queries by evaluating the corresponding OCL expression on the instances of the metamodel that represent the security-design models (or scenarios) under consideration.

The idea of formulating OCL queries on access control policies is not new. Our work is inspired by [1,14], 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 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 models and scenarios to the corresponding instances of this metamodel. As we will see, being explicit about the metamodels and mappings used requires considerable attention to detail. But the payoff is substantial: models and scenarios can then be automatically analyzed in a semantically meaningful way. Second, we show the feasibility of this approach by applying it to a nontrivial example: a security-design modeling language from [4], which combines SecureUML with a component modeling language named ComponentUML. Third, we demonstrate that OCL expressions can be used to formalize and check non-trivial security properties of security-design models. We provide a number of examples that illustrate the expressiveness of this approach for reasoning about properties depending on both the modeled access control policy as well as a snapshot of the dynamic system state. Finally, we provide a tool, SecureMOVA, that implements our approach. All of the examples presented in this paper have been checked using this tool.

Outline. In Section 2, we describe the methodology underlying our approach. Afterwards, in Sections 3 and 4, we describe the different modeling languages employed and their semantics. In Section 5, we explain how scenarios are modeled and, in Section 6, we give examples that illustrate how one can formalize and analyze a wide spectrum of authorization queries 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.

A preliminary version of this paper appeared in [2]. The results reported on there were limited to security-design models without scenarios and based on a more rudimentary version of the SecureMOVA tool.

2. General Approach

Background on (meta)models. A *modeling language* provides a vocabulary (concepts and relations) for building models, as well as a notation to graphically depict them as *diagrams*. Often, the modeling language also provides a vocabulary for describing model instances (also called "scenarios", as in this paper), along with a notation to graphically depict them. In the case of UML, for example, object diagrams represent instances of class models.

Diagrams that depict models (or model instances) must conform to the *metamodel* of the modeling language. A metamodel is a diagram whose elements formalize the concepts and relations provided by the modeling language and whose invariants, usually written in OCL, specify additional well-formedness constraints on models and model instances.

Problem statement: rigorously analyzing security models. Some modeling languages explain the *meaning* of the diagrams using natural language. In this situation, analyzing the models (or the model instances) depicted 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 $[\cdot]$ that associates mathematical structures $[M]$ to well-formed diagrams M.

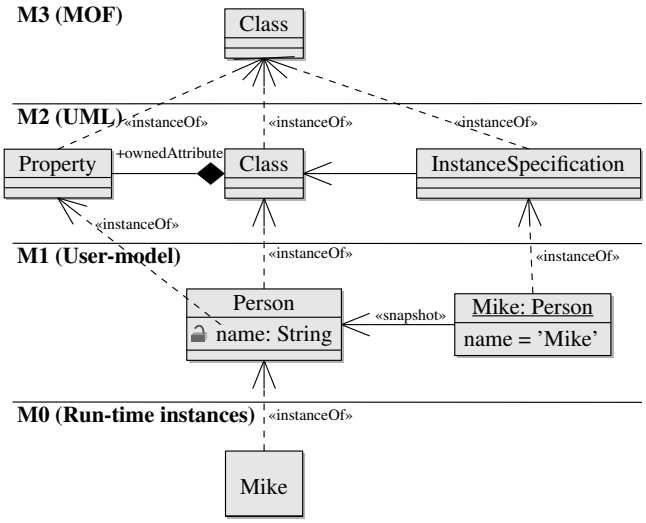
In general, given a modeling language with a formal semantics, one can reason about models by reasoning about their semantics. In the case of a security modeling language $\mathcal{M}$, a security model (or a security model instance) M has a property P (expressed as a formula in some logical language) if and only if $[M] \models P$. While this approach is standard, it either requires deductive machinery for reasoning about the semantics of models (i.e., a semantic embedding [5] and deduction within the relevant semantic domains, which typically cannot be done automatically) 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 (or implemented in M, if M is a security model instance), but in a more familiar setting.

Approach taken. Our approach for analyzing properties of security-design models and their instances reduces deduction to *evaluation*. In a nutshell, we express the desired properties as OCL queries and evaluate these queries on the models (or model instances) under consideration. [2] To

[2] A similar approach has been proposed for UML model metrication [6]: metrics are formalized as OCL queries on the UML metamodel and are evaluated on the models under consideration. This approach appears very general: it can be applied to analyze any model property, independently of the modeling language, as long as the property is expressible in OCL with the types and vocabulary provided by the metamodel.

give a more detailed description of our approach, we first recall the elements involved in using OCL for analyzing model properties. A brief review of OCL itself is provided in Appendix A.

Consider the UML four-layer metamodel hierarchy represented by the following diagram.



As a query language, OCL can be used to analyze an instance of a model at level M0, using the types and vocabulary introduced by the model at level M1. The types correspond to the classes in the model and the vocabulary corresponds to the properties (attributes, roles, and operations) declared for these classes. For example, in OCL one can query the number of persons (Person.allInstances()−>size()) or the name of a person such as Mike (Mike.name). Interestingly, since metamodels are themselves models, OCL can also be used to query the models at level M1, using the types and vocabulary introduced by the metamodel at level M2. For example, in OCL one can query the number of classes in the model (Class.allInstances()) or the number of attributes possessed by a particular class such as Person (Person.ownedAttribute−>size()).

A crucial observation is that, although OCL expressions "talk about" instances of (meta)models, they are in fact interpreted on *representations* of those instances for which the meaning of the new types and vocabulary introduced by the (meta)models is unambiguous. In our example, consider the meaning of the property ownedAttribute: should the attributes "owned" by a class (which is the intended meaning of the property ownedAttribute) include those which are inherited by the class or only those explicitly declared in the class? [3] An adequate representation of UML models as instances of the UML metamodel should provide an unambiguous interpretation for ownedAttribute as well as the rest of the vocabulary introduced by the UML metamodel to "talk about" models. In UML, the standard way of representing instances of (meta)models is through object diagrams (also called "snapshots"). They also have the advantage of providing the information needed to unambiguously interpret OCL expressions written in the context of (meta)models. In what follows, we will denote by *graphical models* the models M that the modeler sees and works with, and by *abstract models* $\overline{M}$ we mean the object diagrams that represent the models M as instances of the metamodel.

Now we can give a more precise description of our approach. Let a security-design modeling language $\mathcal{M}$ be given. In order to analyze properties of security-design models (and model instances) M, we first formalize the desired properties as OCL queries using the types and vocabulary provided by the metamodel of $\mathcal{M}$, and we then evaluate these queries on the corresponding snapshots $\overline{M}$ of the metamodel of $\mathcal{M}$. For this approach to be meaningful, we require that the *mapping* relating graphical models M to abstract models $\overline{M}$, along with the evaluation of OCL expressions, correctly interacts with the interpretation function $[\cdot]$. 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 (for examples, see the related work section), or even wrong. Note that this is particularly prone to happen when one is using modeling languages loaded with syntactic sugar.

Overall, our metamodel-based approach has a number of advantages over more traditional deductive approaches. First, the metamodel of a modeling language is always well-defined and should be known by the modelers. Thus, when analyzing their security-design models, the modelers need not create from scratch, or learn, new (non-standard) concepts and relations. Instead, they can use the same language that they are using for modeling their system and its security policy. Second, OCL is a formal language defined as a standard add-on to UML. As noted in [16], "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 scientists." Thus, the analysis of the models can be carried out by those building them, rather than only by others ("verification engineers") with a strong logical and mathematical background. Last but not least, there are many tools that support working with UML and OCL. In particular, there are tools that can automatically evaluate OCL expressions. Thus, the analysis of the models need not be a time-consuming task, which may require even more effort and ingenuity than the modeling itself. Instead, the modelers can use push-button technologies that are already available in academia and industry. The limitations of our approach are also clear: there may be interesting properties that cannot be naturally expressed using OCL or that cannot be proven by simply evaluating the expressions on concrete security-design models (or scenarios).

[3] In principle, both are possible definitions of the property ownedAttribute. In fact, [13] has to clarify that the reference of the property ownedAttribute "does not include the inherited attributes."

Correctness. Here we expand on the requirements of our approach, in particular how OCL query evaluation must be 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:

$$\begin{array}{ccccc} \text{abstract model} & & \text{graphical model} & & \text{semantic domain} \\ \overline{M} & \leftarrow & M & \rightarrow & [M] \\ \downarrow & & & & \downarrow \\ ev(exp_f, \overline{M}) & & \longrightarrow & & f([M]) \end{array}$$

In this diagram, the downward arrow on the left side denotes the evaluation of the OCL expression exp_f, the result of which is denoted by the function $ev(\cdot,\cdot)$. 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 $[ev(exp_f, \overline{M})] = f([M])$. Roughly speaking, this means that an OCL expression can be correctly used to check a property P if and only if, for arbitrary models (or model instances) M, the result of evaluating this expression on $\overline{M}$ corresponds to the value of the property P in $[M]$.

Rigorously proving this correspondence requires detailed meta-reasoning that involves both the semantics of the underlying formal system, the 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 of the underlying mapping from graphical models to abstract models. This is the approach we have taken in this paper and hence we provide in the following sections precise definitions of the metamodel of our modeling language and of the mappings from graphical models to abstract models. Note that, as explained above, this is already a necessary condition for stating meaningful OCL expressions on models and model instances in the first place.

3. The SecureUML+ComponentUML Language

3.1. *The SecureUML+ComponentUML Metamodel*

SecureUML. SecureUML [4] is a modeling language for formalizing access control requirements that is based on RBAC [7]. In RBAC, permissions specify which roles are allowed to perform given operations. These roles typically represent job functions within an organization. Users are granted permissions by being assigned to the appropriate roles, based on their competencies and responsibilities in the organization. RBAC additionally allows one to organize the roles in a hierarchy, where roles can inherit permissions along the hierarchy. In this way, the security policy can be described in terms of the hierarchical structure of an organization. However, it is not possible to specify policies that depend on dynamic properties of the system state, for example, to allow an operation only during weekdays. SecureUML extends RBAC with *authorization constraints* to overcome this limitation. It formalizes access control decisions that depend on two kinds of information:

(i) Declarative access control decisions that depend on static information, namely the assignments of users and permissions to roles, which we designate as an *RBAC configuration.*

(ii) 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 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. Essentially, it provides a language for modeling *Roles*, *Permissions*, *Actions*, *Resources*, and *Authorization Constraints*, along with their *Assignments*, i.e., which permissions are assigned to which roles, which actions are assigned to which permissions, which resources are assigned to which actions, and which constraints are assigned to which permissions. Notice also that actions can be either *Atomic* or *Composite*. The atomic actions are intended to map directly onto actual operations of the modeled system. The composite actions are used to hierarchically group more lower-level ones and are used to specify permissions for sets of actions.

ComponentUML. The system design modeling language that we consider in this paper, 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 may have *Attributes* and *Methods*. The ComponentUML metamodel is depicted in the (boxed) right-hand part of Figure 2.

Dialect Definition. The dialect metamodel provides the connection between SecureUML and the system design modeling language. The metamodel shown in Figure 2, together with the complete SecureUML metamodel, constitute the *SecureUML+ComponentUML metamodel*.

A dialect metamodel specifies:

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

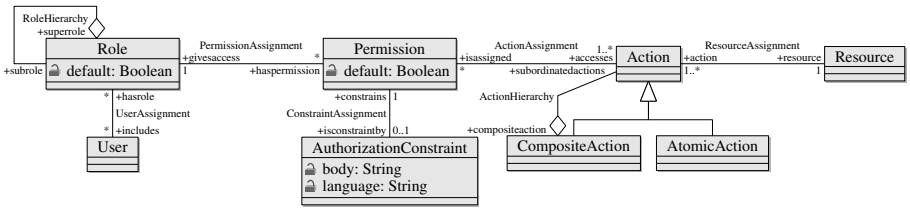


Figure 1. SecureUML Metamodel.

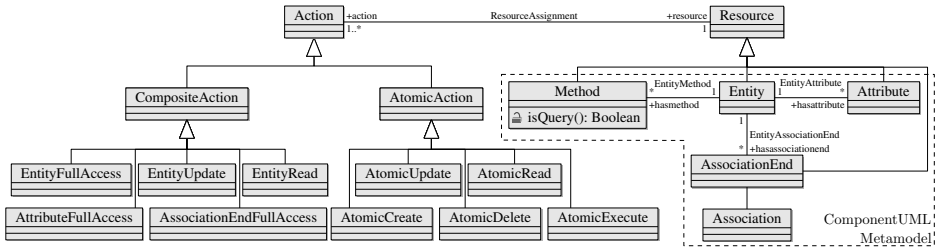


Figure 2. ComponentUML Dialect Metamodel.

(ii) The actions these resource types offer and the hierarchies classifying these actions. This is accomplished in two steps: first, by introducing the different action types as specializations of the classes *CompositeAction* and *AtomicAction*; and second, by using OCL metamodel invariants to constrain which resources are accessible from the actions and which actions are subordinated to the composite actions. The actions offered here are shown in the following table, where underlined actions are composite actions.

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

The complete list of OCL metamodel invariants is given in Appendix B. The list includes, for example, the following invariants, which guarantee that *AttributeFullAccess* actions always act on *Attributes* and that they contain both the read and the update actions upon the corresponding attributes.

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

(iii) The default access control policy for those actions where no explicit permissions are defined (i.e., whether access is allowed or denied by default). This is accomplished by specifying whether the default role has the default permission, where the default role and the default permission are distinguished by having the *default* attribute set to *true*. OCL metamodel invariants (given in Appendix B) therefore include, for example, the following invariants, which guarantee that security models contain a default permission that can only be given to the default role and whose authorization constraint is always satisfied.

```
context Permission
inv existsADefaultPermission:
  self.allInstances()->select(p|p.default)->size() = 1
inv defaultPermissionAssignedToDefaultRole:
  self.default implies
  self.givesaccess->forAll(r|r.default)
inv constraintByTrue:
  self.default implies self.isconstraintby.body = ''true''
```

The default access policy is now specified by the following constraint, saying that the default role has the default permission, i.e., access is allowed by default:

```
context Role
inv defaultAccess:
  self.default implies
  self.haspermission->exists(p|p.default)
```

3.2. *The SecureUML+ComponentUML Models*

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

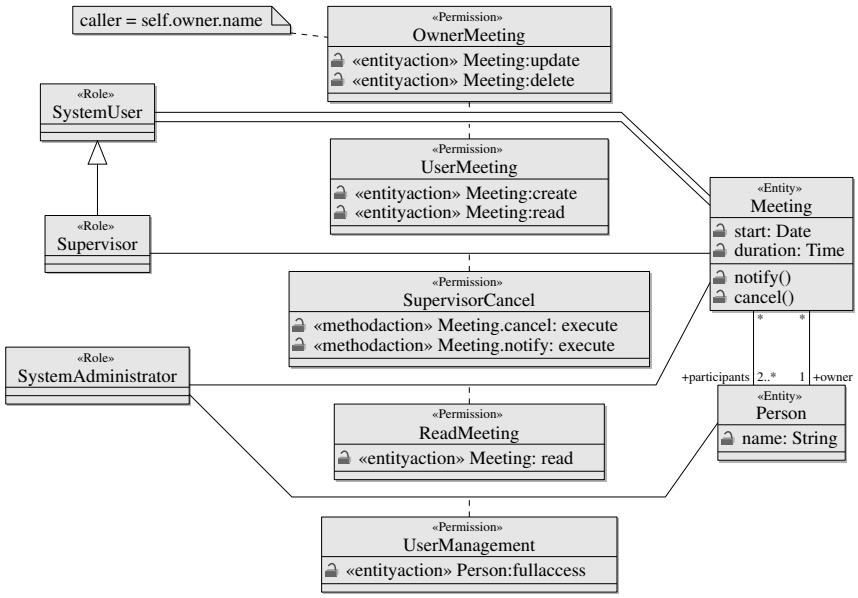


Figure 3. Example Security Policy.

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, and administer the persons in the system.
- A supervisor (but not a system administrator) is also a user of the system.

Figure 3 formalizes this security policy using the UML profile for SecureUML and ComponentUML defined in [4].

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 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 Snapshots of the Metamodel*

Recall that, in our approach, the specification of security properties using OCL directly depends on the mapping from models to snapshots of the metamodel. This is because the expressions formalizing the properties will not be evaluated on the graphical models, but rather on the corresponding abstract models. Of course, the mapping must satisfy the following property: if M is a well-formed graphical model, then $\overline{M}$ is an abstract model that satisfies all the invariants of the metamodel. The complete definition of the mapping from SecureUML+ComponentUML graphical models to the corresponding abstract models is given in Appendix C. To a large extent, this mapping is straightforward: UML model elements with appropriate stereotypes are mapped to snapshots of the corresponding metamodel elements and associations between UML model elements are mapped to links between the snapshots of the corresponding metamodel elements.

In some cases, however, this mapping is less straightforward. In such cases, explicitly giving the mapping is absolutely necessary to avoid ambiguous or incorrect interpre-

tations and to guarantee that the resulting abstract models fulfill the metamodel invariants. This is particularly important when the notation provides the modeler with convenient syntactic sugar. We list below some examples of such subtleties. Let M be a model, then the mapping constructs an $\overline{M}$ that contains (among others) the following elements.

- The mapping creates default objects of type *Role*, *AuthorizationConstraint*, and *Permission*. Note that the default roles, authorization constraints, and permissions are not depicted in M. A default role, for example, is created by the mapping, it is assigned to every user depicted in M, and it is declared a superrole of every role depicted in M. As a result, the following metamodel invariants (which formalize part of the default access control policy defined in SecureUML+ComponentUML) are guaranteed to be fulfilled in $\overline{M}$:

```
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)
```

  Similarly, a default object of type *AuthorizationConstraint* is created by the mapping, with the string *'true'* as the value of its *body* attribute.
- The mapping creates 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*. For example, the mapping creates a *ConstraintAssignment*-link between the default authorization constraint and the default permission, as well as a *PermissionAssignment*-link between the default permission and the default role. As a result, the metamodel invariants *defaultPermissionAssignedToDefaultRole* and *constraintByTrue* (which, as explained in the previous section, formalize part of the default access control policy defined in SecureUML+ComponentUML) are guaranteed to be (partially) fulfilled in $\overline{M}$.
- The mapping creates objects of subtypes of *Action* that correspond to the actions offered by the resources, and links between these objects and the corresponding objects of subtypes of *Resource*. Note that objects for all possible actions are created, not only for those mentioned in the attributes of the permissions depicted in M. As an illustration, consider the model in Figure 3. The objects representing the actions of reading or updating the attribute *name* of the entity *Person*, as well as the composition of these two actions, are created by the mapping, although they are not explicitly referenced by any permission. In addition, the mapping creates *ResourceAssignment*-links between these objects and the object representing the attribute *name*. As a result, the following metamodel invariant, which formalizes the actions defined in SecureUML+ComponentUML on entity attributes, is guaranteed to be fulfilled in $\overline{M}$ for the attribute *name*:

```
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))
```

- The mapping creates links between the objects of type *AtomicAction* and the corresponding objects of type *CompositeAction*. For example, the mapping for the model in Figure 3 creates a *ActionHierarchy*-link between the object of type *CompositeAction* representing the action of reading the entity *Person* and the object of type *AtomicAction* representing the action of reading its attribute *name*. As a result, the following metamodel invariant, (which formalizes part of the hierarchy of actions defined in SecureUML+ComponentUML), is guaranteed to be (partially) fulfilled in $\overline{M}$ for the entity *Person*:

```
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)))
```

Notice that the abstract models contain objects (and links between these objects) that are created by our mapping and therefore do not appear (or have names) in the graphical model. To denote these objects, we use the following conventions.

- Objects representing users, roles, permissions, and entities are named by their name in the graphical model. For example, the object representing the role *SystemUser* as a snapshot of the class *Role* is named *SystemUser* in the abstract model.
- Objects representing actions are named by the name of the resource they act upon, followed by their types. For example, the object representing an atomic update action that acts upon the attribute *start* of the entity *Meeting* is named *MeetingstartAtomicUpdate*.

4. Analyzing SecureUML+ComponentUML Models

In this section, we define OCL operators in the context of the metamodel of SecureUML+ComponentUML that for-

malize different aspects of the access control information contained in the models. We will use these operators 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. Programmatic access control decisions can also be analyzed using these operators as we will discuss in Section 5. 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 on the snapshots of the metamodel corresponding to the models.

To illustrate our approach, we include examples that show the result of evaluating our OCL operators on the security policy modeled in Figure 3. All the examples have been checked with the SecureMOVA tool. In what follows, let SCHEDULER be the snapshot of the metamodel of SecureUML+ComponentUML that corresponds to the model depicted in Figure 3.

4.1. *Semantics*

We recall the semantics of SecureUML+ComponentUML models [4], 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 the 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,

$$\mathcal{S}_{RBAC} = \left\{ \begin{array}{l} \mathit{Users}, \mathit{Roles}, \mathit{Permissions}, \\ \mathit{AtomicActions}, \mathit{Actions} \end{array} \right\},$$

where $\mathit{Actions} \geq_{RBAC} \mathit{AtomicActions}$. Also, $\mathcal{F}_{RBAC} = \emptyset$ and

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

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 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 corresponding metamodel associations *UserAsssignment*, *PermissionAssignment*, and *ActionAssignment*. Additionally, we define the partial orders $\geq_{\mathit{Roles}}$ and $\geq_{\mathit{Actions}}$ on the sets of roles and actions, respectively. $\geq_{\mathit{Roles}}$ is given by the reflexive closure of the metamodel aggregation *RoleHierarchy* on *Role*, and we write subroles (roles with additional privileges) on the left (larger) side of the $\geq$-symbol. $\geq_{\mathit{Actions}}$ is given by the reflexive closure of the composition hierarchy on actions, defined by the metamodel aggregation *ActionHierarchy*. We write $a_1 \geq_{\mathit{Actions}} a_2$, if a_2 is a subordinated action of a_1. These relations are partial orders since aggregations in UML are transitive and antisymmetric by definition.

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

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)

Next, we introduce the formulas

$$\phi_{\mathit{User}}(u, p) := \exists r, r' \in \mathit{Roles} : \mathit{UA}(u, r) \wedge r \geq_{\mathit{Roles}} r' \wedge \mathit{PA}(r', p)\ ,$$

and

$$\phi_{\mathit{Action}}(p, a) := \exists a' \in \mathit{Actions} : \mathit{AA}(p, a') \wedge a' \geq_{\mathit{Actions}} a\ .$$

Here, $\phi_{\mathit{User}}(u, p)$ denotes that u has permission p and $\phi_{\mathit{Action}}(p, a)$ denotes that p is a permission for action a.

The access control semantics is now given by the formula $\phi_{RBAC}(u, a)$, which expresses that a user u has a permission to perform the action a.

$$\phi_{RBAC}(u, a) := \exists p \in \mathit{Permissions} : \phi_{\mathit{User}}(u, p) \wedge \phi_{\mathit{Action}}(p, a)\ .$$

4.2. *Analysis Operations*

In this section, we define different OCL query operations that are useful for analyzing security properties of models formalized using SecureUML+ComponentUML.

To analyze the relation $\geq_{\mathit{Roles}}$ in the Σ_{RBAC}-structure $\Im_{RBAC}$ defined by a model M, we define the operation Role::superrolePlus():Set(Role) 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.superrole->exists(r|rs->excludes(r))
  then self.superrolePlusOnSet(rs
       ->union(rs.superrole)->asSet())
  else rs->including(self)
```

```
  endif
```

In our example, Supervisor.superrolePlus() evaluates to Set{defaultRole, SystemUser, Supervisor} on SCHEDULER.

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()
```

In our example, Supervisor.allPermissions() evaluates to Set{defaultPermission, OwnerMeeting, UserMeeting, SupervisorCancel} on SCHEDULER.

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

To analyze the relation $\geq_{Actions}$ in the Σ_{RBAC}-structure $\Im_{RBAC}$ defined by a model M, we define the operation 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
```

In our example, MeetingEntityUpdate.subactionPlus() evaluates to Set{MeetingstartAtomicUpdate, MeetingdurationAtomicUpdate, MeetingcancelAtomicExecute, MeetingnotifyAtomicExecute, MeetingownerAtomicUpdate, MeetingparticipantsAtomicUpdate} on SCHEDULER.

Similarly, we define 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()
```

In our example, OwnerMeeting.allActions() evaluates to Set{ MeetingAtomicDelete, MeetingstartAtomicUpdate, MeetingdurationAtomicUpdate, MeetingcancelAtomicExecute, MeetingnotifyAtomicExecute, MeetingownerAtomicUpdate, MeetingparticipantsAtomicUpdate} on SCHEDULER.

Conversely, we define 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: 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*, Table 1 shows the 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_1.superrolePlus()->includes(r_2)
	r_2.subrolePlus()->includes(r_1)
$\exists r_2 \in Roles.$	r_1.allPermissions()->includes(p)
$r_1 \geq_{Roles} r_2 \wedge PA(r_2, p)$	p.allRoles()->includes(r_1)
$a_1 \geq_{Actions} a_2$	a_1.subactionPlus()->includes(a_2)
	a_2.compactionPlus()->includes(a_1)
$\phi_{User}(u, p)$	u.hasrole.allPermissions()->includes(p)
	p.allRoles().includes->includes(u)
$\phi_{Action}(p, a)$	p.allActions()->includes(a)
	a.allAssignedPermisssions()->includes(p)
$\phi_{RBAC}(u, a)$	u.allAllowedActions()->includes(a)

Table 1
Correspondence between Semantics and OCL Evaluation.

5. Analyzing Scenarios

To analyze the effect of authorization constraints on a SecureUML policy, we must be able to formalize and reason about instances of the ComponentUML model. Hence, the user needs to be able to specify system states, which we will call *security scenarios* in the following.

In a security scenario, the user specifies both the instance of the ComponentUML part of the security-design model (in the manner of a UML object diagram), as well as role assignments for the users in the scenario. We assume here that SecureUML users are represented as ComponentUML entities. The example scenario in Figure 4 contains two users, *Bob* and *Alice* with roles *SystemUser* and *Supervisor* respectively. It also contains one meeting, *Kick-off*, whose owner is *Alice*.

Recall that our general approach for analyzing models is based on evaluating OCL expressions on snapshots of the metamodel. This means, we first must define the metamodel that formalizes the concepts and relations which are modeled by the security scenarios of security-design models. In a second step, we must define how to map the security-design model together with the security scenario to a snapshot of this new metamodel.

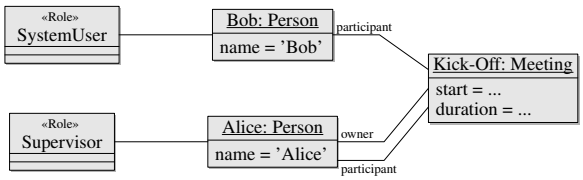


Figure 4. Example Scenario

5.1. *Metamodel Combination*

Our goal is to reason about the static RBAC configuration given in the security-design model together with the effects of authorization constraints in the given security scenario. Hence, the metamodel in which our analysis questions can be formalized as OCL expressions must provide the vocabulary for both. The idea thus is to merge the SecureUML metamodel with the ComponentUML model that is part of the security-design model (i.e., the design model) so that queries involving the validity of authorization constraints can be formulated using OCL expressions on snapshots of this combined metamodel.

This merging is more than a simple union of both models (the SecureUML metamodel and the ComponentUML model). Connections between them must be made in order to formulate queries involving, for example, both role assignments and system state. To illustrate this metamodel combination, Figure 5 shows the result of the merging for the case of the security-design model in Figure 3 and the scenario in Figure 4. In particular:

(i) The ComponentUML entities are added to the SecureUML+ComponentUML metamodel and are given a common supertype *EntityInstance*, which is associated to *Entity*. This reflects the fact that in a scenario a person *Alice* is an instance of the class *Person* but, as a concrete resource, she also has a type *Person*, which is an instance of *Entity*. This is similar to how the relationship between objects and classes is modeled in the UML metamodel. Only here, the metamodel is not fixed, but is extended by the entities depicted in the security design model (in our example, *Person* and *Meeting*).

To guarantee that entity instances are associated to their corresponding entities, for each pair of distinct entities A and B depicted in the security-design model (in our example, *Person* and *Meeting*) the following invariants are added to the SecureUML+ComponentUML metamodel:

```
context A inv:
  A.allInstances()->forall(x,y| x.entity = y.entity)

context A inv:
  B.allInstances()->forall(b| b.entity <> self.entity)
```

(ii) An inheritance relationship from the ComponentUML entity representing users to the class *User* is added to the SecureUML+ComponentUML metamodel. This reflects the fact that, in a scenario, a person *Alice* is an instance of the class *Person* but, as a user, she can also be a *caller* in an authorization constraint. In general, this user-entity will be implicitly defined by the actual role-assignments in the scenario: The entity whose instances are associated to roles obviously represents users (in our example, the entity *Person*).

(iii) A class *ActionInstance*, with associations both to *Action* and *EntityInstance*, is added to the SecureUML+ComponentUML metamodel. This reflects the fact that, in a scenario, the action of reading *Alice*'s *name* has a type, namely, the type represented by the atomic action *PersonnameAtomicRead* and it also refers to an instance of an entity, namely, the person *Alice*. To guarantee that action instances are created for each pair of compatible atomic action and entity instance, and that they are associated to their corresponding action and entity instances, the following invariants are added to the SecureUML+ComponentUML metamodel:

```
context Resource::root():Entity post:
  if self.oclIsTypeOf(Entity) then self
  else if self.oclIsTypeOf(Method) then self.entity
  else if self.oclIsTypeOf(Attribute) then self.entity
  else if self.oclIsTypeOf(AssociationEnd)
          then self.entity
  else OclUndefined
  endif endif endif endif

context AtomicAction
inv: self.actionInstance->size() =
     self.resource.root().entityInstance->size()
inv: self.resource.root().entityInstance->forall(ei|
       self.actionInstance->exists(ai|
         ai.resourceInstance = ei))

context ActionInstance inv:
  self.action.resource.root() =
  self.resourceInstance.entity
```

(iv) Finally, a method

```
AuthorizationConstraint::evaluate(caller:User,
    self:EntityInstance):Boolean
```

is added to *AuthorizationConstraint* in the SecureUML+ComponentUML metamodel, with the

following semantics: it evaluates the constraint on the given scenario, using the arguments *caller* and *self* to instantiate the parameters caller and self in the constraint.

5.2. *Scenario Mapping*

Given the combined metamodel, a security-design model together with a security scenario can be mapped to a snapshot of this metamodel. The complete definition is given in Appendix C. For the most part, this mapping is straightfoward: the SecureUML policy in the security-design model is mapped as described in Section 3.3, and the object diagram in the security scenario (i.e., the instance of the ComponentUML part in the security-design model) is mapped as expected. There are, however, some subtleties that we list below. We use as an example the security-design model in Figure 3 together with the scenario in Figure 4.

- Links are created to connect each instance of *EntityInstance* with the corresponding instance of *Entity*. In our example, the mapping creates a link between the object *Alice* of the class *EntityInstance* (that represents the person *Alice* depicted in the security scenario), and the object *Person* of the class *Entity* (that represents the entity *Person*). This link reflects the type of *Alice* as a resource, namely, a *Person*.
- Objects of the type *ActionInstance* are created for each instance of *AtomicAction* and each instance of *Entity* such that the latter is the (root) resource for the former in the security-design model. Also, links are created to connect these instances of *ActionInstance* with the corresponding instances of *AtomicAction* and *EntityInstance*. In our example, an object *Kick-OffstartAtomicRead* of the class *ActionInstance* is created, and it is linked both to the object *MeetingstartAtomicRead* of the class *AtomicRead* (that represents the action of reading the value of the attribute *start* of the entity *Meeting*) and to the object *Kick-Off* of the class *Meeting* (that represents the meeting *Kick-Off* depicted in the security scenario). These links reflect, respectively, the type of *Kick-OffstartAtomicRead* as an action, namely, the action *MeetingstartAtomicRead*, and the resource upon which it acts, namely, the meeting *Kick-Off*.

5.3. *Scenario Analysis Operations*

Let us first recall here the semantics of programmatic access control decisions in SecureUML+ComponentUML models, with respect to which we claim that our OCL-operations are correct. Let Σ_{ST} be the order-sorted signature corresponding to the ComponentUML model and let $\mathfrak{S}_{ST}$ be the Σ_{ST}-structure defined by the security scenario. For the formal definition of Σ_{ST} and $\mathfrak{S}_{ST}$, we refer to [4]. Basically, Σ_{ST} contains a sort for each entity and a function symbol for each attribute or association end of an entity in the ComponentUML model. The definition of $\mathfrak{S}_{ST}$ from the security scenario is the straightforward one. Let $\Sigma_{AC} = \Sigma_{RBAC} \cup \Sigma_{ST}$, and $\mathfrak{S}_{AC} = \langle \mathfrak{S}_{ST}, \Im_{RBAC} \rangle$.

Given the above, a user u is allowed to perform the action a if and only if

$$\mathfrak{S}_{AC} \models \bigvee_{p \in Permissions} \phi_{User}(u,p) \wedge \phi_{Action}(p,a) \wedge \phi^p_{ST}(u) \ ,$$

where $\phi^p_{ST}(u)$ denotes the authorization constraint attached to permission p.

Equivalently, a user u is allowed to perform the action a if and only if

$$\mathfrak{S}_{ST} \models \bigvee_{p \in \mathrm{UAP}(u,a)} \phi^p_{ST}(u) \ .$$

Here, $\mathrm{UAP}(u,a)$ is a function returning the set of permissions that u has for the action a:

$$\mathrm{UAP}(u,a) := \{p \in Permissions \mid \phi_{User}(u,p) \wedge \phi_{Action}(p,a)\} \ .$$

Thus, access is granted if and only if the user u has a permission p for which the corresponding authorization constraint is valid in $\mathfrak{S}_{ST}$.

In what follows, let SCHEDULER+ be the snapshot of the metamodel that corresponds to the scenario depicted in Figure 4. We start by defining an operation User::allAuthConstUser(a:Action):Set(AuthorizationConstraint) that returns the list of authorization constraints for a given user-action pair.

```
context User::allAuthConstUser(a:Action)
            :Set(AuthorizationConstraint) body:
  self.hasrole.superrolePlus().allAuthConstRole(a)

context Role::allAuthConstRole(a:Action)
            :Set(AuthorizationConstraint) body:
  self.permissionPlus(a).isconstraintby
```

Given this, we can now define the most basic analysis operation: can a given user perform a given action instance in the current scenario? This is the operation User::isAllowed(a:ActionInstance):Boolean defined below.

```
context User::isAllowed(a:ActionInstance):Boolean body:
  self.allAuthConstUser(a.action)->exists(au|
     au.evaluate(self,a.resourceInstance))
```

In our example, Alice.isAllowed(Kick-OffAtomicDelete) evaluates to true on SCHEDULER+. We can also calculate the allowed action instances for a given user and the set of users that are allowed to perform a given action instance.

```
context User::allAllowedActionInstances()
            :Set(ActionInstance) body:
    self.allAllowedActions().actionInstance
    ->select(ai|(self.isAllowed(ai))
```

In our example, Bob.allAllowedActionInstances() evaluates to Set{ Kick-OffAtomicCreat, Kick-OffstartAtomicRead, Kick-

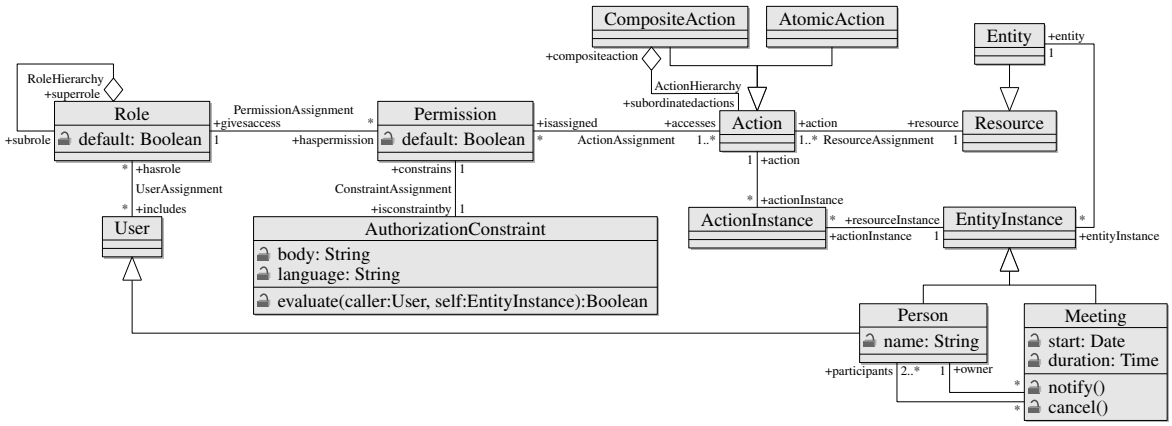


Figure 5. Combined Metamodel

OffdurationAtomicRead, Kick-OffownerAtomicRead, Kick-OffparticipantsAtomicRead, e} on SCHEDULER+.

context ActionInstance::allUsers():Set(User) **body**:
self.action.allAssignedUsers−>select(u|u.isAllowed(self))

context Action::allAssignedUsers():Set(User) **body**:
self.allAssignedRoles.includes

In our example, Kick-OffAtomicDelete.allUsers() evaluates to Set{Alice} on SCHEDULER+.

Remark: Let $\mathfrak{S}_{AC}$ be the Σ_{ST}-structure defined by a model M together with a security scenario. Then, for any u in *Users* and a in *Actions*, and for any action instance ai of a, Table 2 shows the correspondence between satisfaction in $\mathfrak{S}_{AC}$ and evaluation of OCL expressions in $\overline{M}$.

is satisfied in $\mathfrak{S}_{AC}$	evaluates to true in $\overline{M}$
$\exists p \in Permissions.$ $p \in \mathrm{UAP}(u,a)$	u.allAuthConstUser(a)−>includes(ϕ^p_{ST})
$\exists p \in Permissions.$ $p \in \mathrm{UAP}(u,a) \wedge \phi^p_{ST}$	u.isAllowed(ai) u.allAllowedActionInstances()−>includes(ai) ai.allUsers()−>includes(u)

Table 2
Correspondence between Semantics and OCL Evaluation.

6. Analysis Examples

In this section, we give examples that illustrate how one can analyze SecureUML+ComponentUML models or scenarios M using the OCL operations defined in the previous sections. The properties to be analyzed are formalized as queries on objects in $\overline{M}$, possibly with additional arguments referring to the objects in $\overline{M}$. We also show the results of the queries on the security policy modeled in Figure 3 and the scenario modeled in Figure 4. All the examples have been checked with the SecureMOVA tool.

6.1. *Security policies*

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

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

In our example, SystemAdministrator.allAtomics() evaluates to Set{MeetingstartAtomicRead, MeetingdurationAtomicRead, MeetingownerAtomicRead, MeetingparticipantsAtomicRead, PersonnameAtomicUpdate, PersonmeetingAtomicUpdate, PersoneventsAtomicUpdate, PersonnameAtomicRead, PersonmeetingAtomicRead, PersoneventsAtomicRead, PersonAtomicCreate, PersonAtomicDelete} on SCHEDULER.

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

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

In our example, MeetingAtomicDelete.allAssignedRoles() evaluates to Set{Supervisor, SystemUser} on SCHEDULER.

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))

In our example, Supervisor.allAuthConst(MeetingCancelAtomicExecute) evaluates to Set{"self.name.owner = caller.name", "true"} on SCHEDULER.

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())

In our example, duplicateRoles() evaluates to true on SCHEDULER.

Example: 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()))
```

In our example, PersoneventsAtomicRead.minimumRole() evaluates to Set{SystemAdministrator} on SCHEDULER.

Example: Do two permissions overlap?

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

In our example, OwnerMeeting.overlapsWith(SupervisorCancel) evaluates to true on SCHEDULER.

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())))
```

In our example, existOverlapping() evaluates to true on SCHEDULER.

Example: 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)))
```

In our example, accessAll() evaluates to true on SCHEDULER.

6.2. *Security Scenarios*

Many of the analysis examples above can be expressed in terms of action instances instead of actions. In the following, we provide several examples of this.

Example: Given an action on a concrete resource, which roles are to be assigned to a given user so as to allow her to perform the action in the given scenario?

```
context User::allRolesToPerform(ai:ActionInstance)
              :Set(Role) body:
    Role.allInstances()->select(r|
      r.allAuthConstRole(ai.action)->exists(au|
        au.evaluate(self, ai.resourceInstance)))
```

In our example, Bob.allRolesToPerform(Kick-OffstartAtomicUpdate) and Bob.allRolesToPerform(Kick-OffcancelAtomicExecute) evaluate, respectively, to Set{} and Set{Supervisor} on SCHEDULER+.

Example: Are there actions on concrete resources that every user can perform in the given scenario?

```
context ActionInstance::accessAllUsers():Boolean body:
    ActionInstance.allInstances()->exists(ai|
      User.allInstances()->forAll(u|u.isAllowed(ai)))
```

In our example, accessAllUsers() evaluates to true on SCHEDULER+.

7. The SecureMOVA Tool

As [14] observed, although there are different 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. In particular, tools are needed that can be easily applied by policy designers without much additional training. In response to this need, [14] 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 SecureMOVA, a modeling and analysis tool for SecureUML+ComponentUML that is directly based on the results presented in this paper.

The SecureMOVA tool is an extension of the MOVA tool. MOVA itself is a modeling and validation tool for UML and OCL. MOVA provides facilities for drawing UML class and object diagrams, expressing and evaluating OCL constraints and queries, defining new OCL operations, and evaluating OCL metrics. The OCL editor includes a model-based syntax-guided facility where, to write an expression, the user selects patterns from lists, generated at runtime. The patterns presented to the user depend on the type of the expression and the types and vocabulary introduced in the model under consideration.

The SecureMOVA tool extends MOVA to support the automated analysis of security-design models. In particular, it allow users to draw SecureUML+ComponentUML models and scenarios, to write OCL authorization constraints and assign them to permissions, and to express and evaluate OCL queries on security-design models and scenarios. It shares with MOVA its model-based syntax-guided OCL editor.

Figures 6 and 7 show three screenshots of the SecureMOVA tool. First, Figure 6 shows how the security policy in Figure 3 can be modeled in SecureMOVA. Then, Figure 7 (left) illustrates the use of the OCL editor for querying the example scenario in Figure 4 with one of the queries introduced in Section 6, namely, Bob.AllRolesToPerform(Kick-OffCancelAtomicExecute). Finally, Figure 7 (right) shows the answer to this query after evaluating it on the given scenario. The tool is publicly available at `maude.sip.ucm.es/mova`, along with a tutorial and the security-design model and scenario presented in this paper.

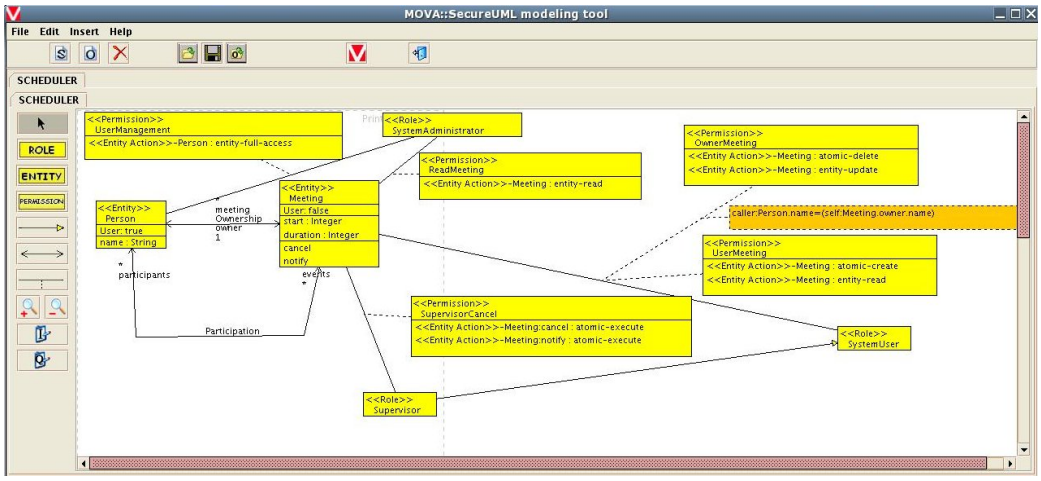


Figure 6. A SecureMOVA security-design model.

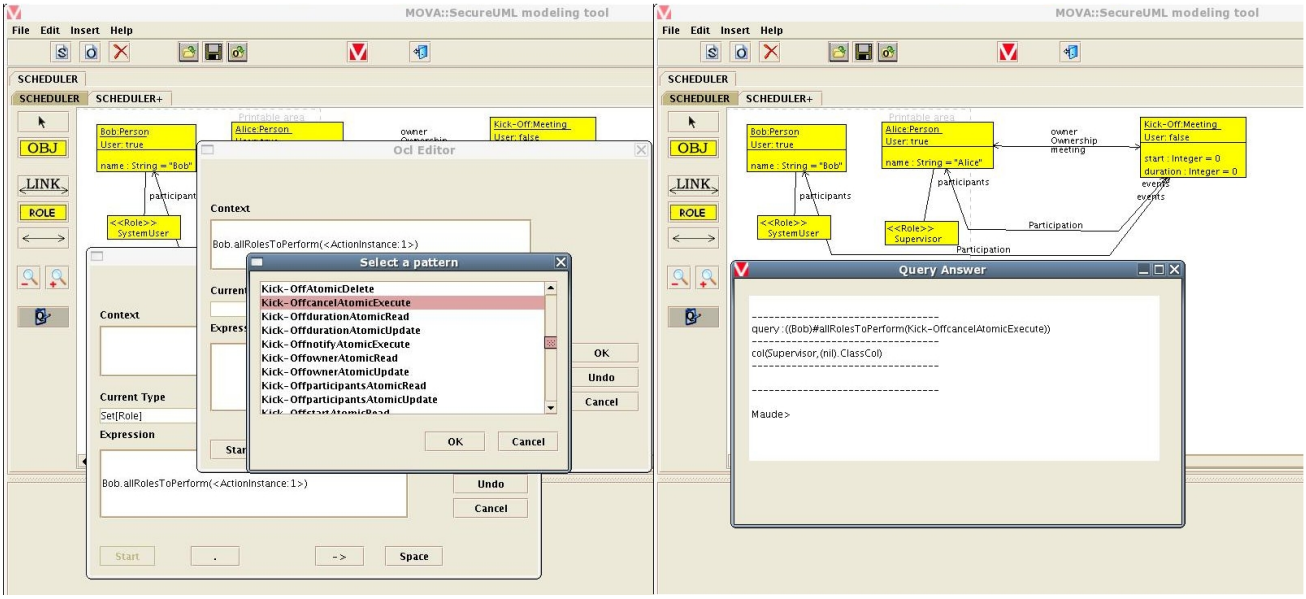


Figure 7. A SecureMOVA OCL pattern-selection (left) and OCL evaluation (right).

8. Conclusion

Related Work. As mentioned in the introduction, our work was inspired by [1], who first explored the use of OCL for querying RBAC policies (see also [14,15]). We discuss two of the main differences here. First, a distinct characteristic of our work is that we spell out and follow a precise methodology that 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 and scenarios to the corresponding snapshots 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 no individual can be assigned to both roles. In [1,14,15] 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 all role assignments explicitly depicted as well as those implicitly assigned to users via the role hierarchy. The actual meaning of the association-end *role* depends, of course, on the mapping between models and the corresponding snapshots of the metamodel. Since the precise definition of this mapping is not given in [1,14,15], readers (and tool users) must speculate on

the meaning of such expressions. (Note that if the mapping used in [1,14,15] is the "straightforward" one, then 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 under the specified role hierarchy.

A second difference concerns the fact that the SecureUML modeling language includes the possibility of constraining permissions with authorization constraints, given by OCL formulas. These constraints restrict the permissions to those system states satisfying the constraints. In our approach, these states can be formalized within models as (security) scenarios and we can automatically determine the satisfaction of authorization constraints with respect to given scenarios. As explained in Section 5, this allows us to go beyond querying properties about a system's static RBAC configuration by allowing queries to refer to, and answers to depend on, the system state, i.e., the current instance of the design model (the ComponentUML part of the security-design model). We have given examples of this in Section 6, where queries combine aspects of both declarative and programmatic access control. For example, "are there actions on concrete resources that every user can perform in a given scenario?"

Generality of Approach. We comment now on the generality of our approach and the effort needed to use it in other settings, for example, when employing other (domain-specific) design modeling languages. Consider, for example, the metamodel constraints shown in Appendix B. Some of these constraints only depend on the fact that the underlying access control model is a variant of RBAC, for example, constraints describing the general structure of role hierarchies. The majority, however, explicitly concern the structure of the design modeling language, for example, the action hierarchy. Transferring our approach to another design modeling language requires that we also define the corresponding metamodel constraints for the new language. Analogous considerations hold for the mapping from graphical to abstract models, described in Appendix C. This is not, however, a peculiarity of our approach. The definition of the dialect for the design-modeling language must include this information anyway so that the meaning of the language can be understood. Note that the metamodel constraints in Section 5.1 not only depend on the design modeling language, but also on the design model. However, these constraints are automatically generated from the design model. This generation scheme again only depends on the design modeling language. Interestingly, the analysis examples given in Section 6 do *not* depend on the design modeling language or on the design model. One could therefore imagine a library of such constraints that could, for example, be used to automatically suggest possible flaws in the modeled security policy.

Future Work. The use of scenarios provides support for handling queries involving system state where we can evaluate queries with respect to a given scenario. Nevertheless, it would be attractive to support queries about the existence of states satisfying constraints or queries where the states themselves are existentially quantified. An example of the latter 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 the consequences of OCL formulas and this involves theorem proving as opposed to 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. We showed in [4] how one can combine SecureUML with different modeling languages (i.e., ComponentUML and ControllerUML [3]) 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?" This is a query about the existence (reachability) of states and answering it would again require support for theorem proving or other forms of deduction such as constraint solving or state search.

References

[1] G. J. Ahn, 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, IEEE Computer Society, Washington, DC, USA, 2001.

[2] D. Basin, M. Clavel, J. Doser, M. Egea, A metamodel-based approach for analyzing security-design models, in: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), vol. 4735 of Lecture Notes in Computer Science, Springer-Verlag, 2007.

[3] D. Basin, J. Doser, T. Lodderstedt, Model driven security for process-oriented systems, in: Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT 2003), ACM Press, 2003.

[4] D. A. Basin, J. Doser, T. Lodderstedt, Model driven security: From UML models to access control infrastructures., ACM Transactions on Software Engineering and Methodology 15 (1) (2006) 39–91.
[5] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, J. V. 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, North-Holland, 1992.
[6] F. B. e Abreu, Using OCL to formalize object oriented metrics definitions, Tech. Rep. ES007/2001, FCT/UNL and INESC, Portugal, available at `http://ctp.di.fct.unl.pt/QUASAR/Resources/Papers/others/MOOD_OCL.pdf` (June 2001).
[7] D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, R. Chandramouli, Proposed NIST standard for Role-Based Access Control, ACM Transactions on Information and System Security 4 (3) (2001) 224–274.
[8] G. Georg, I. Ray, R. France, Using aspects to design a secure system, in: ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, Washington, DC, USA, 2002.
[9] J. Jürjens, Towards development of secure systems using UMLsec, in: H. Hussmann (ed.), Fundamental Approaches to Software Engineering (FASE/ETAPS 2001), vol. 2029 of Lecture Notes in Computer Science, Springer-Verlag, 2001.
[10] J. Jürjens, UMLsec: Extending UML for secure systems development, in: J.-M. Jézéquel, H. Hussmann, S. Cook (eds.), UML 2002 — The Unified Modeling Language, vol. 2460 of Lecture Notes in Computer Science, Springer-Verlag, 2002.
[11] A. Kleppe, W. Bast, J. B. Warmer, A. Watson, MDA Explained: The Model Driven Architecture–Practice and Promise, Addison-Wesley, 2003.
[12] Object Management Group, UML 2.0 OCL Specification, OMG document available at `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14` (2003).
[13] Object Management Group, Unified Modeling Language: Infrastracture, Version 2.1.1, OMG document available at `http://www.omg.org/cgi-bin/doc?formal/07-02-04` (2007).
[14] K. Sohr, G. J. Ahn, M. Gogolla, L. Migge, Specification and validation of authorisation constraints using UML and OCL., in: Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005), vol. 3679 of Lecture Notes in Computer Science, Springer-Verlag, 2005.
[15] H. Wang, Y. Zhang, J. Cao, J. Yang, Specifying Role-Based Access Constraints with Object Constraint Language, in: Proceedings of the 6th Asia-Pacific Web Conference (APWeb 2004), vol. 3007 of Lecture Notes in Computer Science, Springer-Verlag, 2004.
[16] J. Warmer, A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, 2nd ed., Addison-Wesley, 2003.

Appendix A. Background on OCL

OCL has played a central role in our work for formalizing both constraints and queries on models. In this section, we briefly summarize relevant aspects of OCL.

The Object Constraint Language (OCL) [12] is a typed language, with an object-oriented, textual notation, for writing constraints within, or queries on, UML models. The language includes predefined types like Boolean, Integer, and String, with standard operators like not and or over Boolean, +, and ∗ over Integer, and substr and concat over String. For example, 2 + 5 and not(2 + 5 = 6) are OCL expressions of type Integer and Boolean, respectively. The language also provides operators for generating collection types from more basic types, along with standard operations on collections like union, includes, or size. For example, Set(Integer) is the type of sets of integers and 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, operate 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 evaluates to true if and only if each element of the set $\{1, 4, 6\}$ is greater than 7. As another example, collect applies an OCL expression to each element in the collection and returns the union of these results: Set{1, 4, 6}−>collect(i|i + 1) is an expression of type Set(Integer) that denotes the set of integers that result from adding 1 to each of the elements of the set $\{1, 4, 6\}$.

The OCL language is open in the sense that it is parametric. Expressions are written in the context of 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 this class 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 provides a convenient shorthand notation for navigating over multiple association ends: Whenever a property call (attribute, operation, or association end call) is applied to a collection, it will be interpreted as a collect over the members of the collection with the specified property. In the context of Meeting for example, self.participants.name is shorthand for self.participants−>collect(p|p.name) and refers to the collection of the names of the participants of the meeting.

OCL also provides access to the value of certain properties of the classes themselves using the dot notation. For example, the expression A.allInstances() denotes the set of all objects of the class A.

Appendix B. 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 guarantees that the role hierarchy is acyclic.

```
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)
inv constraintByTrue:
    self.default implies self.isconstraintby.body = ''true''

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 guarantee 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)))
```

Appendix C. The Mapping from Graphical to Abstract Models

In the following, we give a complete, albeit informal, definition of the mapping from SecureUML+ComponentUML graphical models and scenarios to the corresponding abstract models.

From security-policy models to abstract models.

- 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) an ActionHierarchy-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.
- 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-link 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 $ae(\overline{m})$.

From security-policy scenarios to abstract models.
- For each object ei of an entity e, insert (i) an object $\overline{ei}$ of the class $\overline{e}$; (ii) an Entity−EntityInstance link between $\overline{e}$ and $\overline{ei}$; (iii) an object $aci(\overline{ei})$ of the class AtomicActionInstance; (iv) an object $adi(\overline{ei})$ of the class AtomicActionInstance; (v) EntityInstance−ActionInstance links between $\overline{ei}$ and $aci(\overline{ei})$, and $\overline{ei}$ and $adi(\overline{ei})$; (vi) Action−ActionInstance links between $ac(\overline{e})$ and $aci(\overline{ei})$, and $ad(\overline{e})$ and $adi(\overline{ei})$;
- For each attribute a of an entity e, and each object ei of this entity, insert (i) an object $ari(\overline{a})$ of the class ActionInstance; (ii) an object $aui(\overline{a})$ of the class ActionInstance; (iii) Action−ActionInstance links between $ar(\overline{a})$ and $ari(\overline{a})$, and $au(\overline{a})$ and $aui(\overline{a})$; (iv) ActionInstance−EntityInstance links between $ari(\overline{a})$ and $\overline{ei}$, and $aui(\overline{a})$ and $\overline{ei}$.
- For each method m of an entity e and each object ei of this entity, insert (i) an object $aei(\overline{m})$ of the class ActionInstance; (ii) an Action−ActionInstance link between $ae(\overline{m})$ and $aei(\overline{m})$; (iii) an ActionInstance−EntityInstance link between $aei(\overline{m})$ and $\overline{ei}$.
- For each association-end d of an entity e, and for each object ei of this entity, insert (i) an object $ari(\overline{d})$ of the class ActionInstance; (ii) an object $aui(\overline{d})$ of the class ActionInstance; (iii) Action−ActionInstance links between $ar(\overline{d})$ and $ari(\overline{d})$, and $au(\overline{d})$ and $aui(\overline{d})$. (iv) ActionInstance−EntityInstance links between $ari(\overline{d})$ and $\overline{ei}$, and $aui(\overline{d})$ and $\overline{ei}$.