Automated Analysis of Security-Design Models

David Basin\textsuperscript{a,}\textsuperscript{*} Manuel Clavel\textsuperscript{b,c,1} Jürgen Doser\textsuperscript{a} Marina Egea\textsuperscript{c,1}

\textsuperscript{a}Information Security Group, ETH Zurich, Zürich (Switzerland)
\textsuperscript{b}IMDEA Software Institute, Madrid (Spain)
\textsuperscript{c}Computer Science Department, Universidad Complutense de Madrid, Madrid (Spain)

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 a 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, representing 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.

Our use of OCL results in 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 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. 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 and notation for describing model instances, also called scenarios. 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 [·] 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 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] |= 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 give a more detailed description of our approach, we first recall

\(^2\) A similar approach has been proposed for UML model metricalation [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.
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 (name = "Mike"). 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”). The object diagrams also provide 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 of the metamodel. 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 are properties that cannot be naturally expressed using OCL or that cannot be proven by simply evaluating the expressions on concrete security-design models and scenarios.

3 In principle, both are possible definitions of the property ownedAttribute. In fact, [13] must 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
SecureUML+ComponentUML metamodel. 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.

(ii) The actions these resource types offer and the hierarchies classifying these actions. This is accomplished in two steps. First, we introduce the different ac-
tion types as specializations of the classes CompositeAction and AtomicAction. Second, we use 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.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>create, read, update, delete, full access</td>
</tr>
<tr>
<td>Attribute</td>
<td>read, update, full access</td>
</tr>
<tr>
<td>Method</td>
<td>execute</td>
</tr>
<tr>
<td>AssociationEnd</td>
<td>read, update, full access</td>
</tr>
</tbody>
</table>

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, stating 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 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 system’s users.
– 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 the permission. The type of the permission attribute specifies the action (e.g., read, update, or full access) that is permitted by the 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 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 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 essential to avoid ambiguous or incorrect interpretations and to guarantee that the resulting abstract models fulfill the metamodel invariants. This is particularly important when the notation provides the modeler with syntactic sugar. We list below some examples of such subtleties. Let $M$ be a
model, then the mapping constructs an $\mathcal{M}$ 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 $\mathcal{M}$:

  ```
  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 $\mathcal{M}$.

- 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 $\mathcal{M}$ 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 an 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 $\mathcal{M}$ 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).hasmethod
      .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 formalize 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 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 $\Sigma_{RBAC} = (S_{RBAC}, F_{RBAC}, R_{RBAC}, P_{RBAC})$ be the order-sorted signature that defines the type of structures specifying role-based access control configurations. Here $S_{RBAC}$ is a set of sorts, $\geq_{RBAC}$ is a partial order on $S_{RBAC}$, $F_{RBAC}$ is a sorted set of function symbols, and $P_{RBAC}$ is a sorted set of predicate symbols. In detail,

$S_{RBAC} = \{ \text{Users, Roles, Permissions, AtomicActions, Actions} \}$,

where $Actions \geq_{RBAC} AtomicActions$. Also, $F_{RBAC} = \emptyset$ and

$P_{RBAC} = \{ \geq_{Roles} : Roles \times Roles, \geq_{Actions} : Actions \times Actions, UA : Users \times Roles, PA : Roles \times Permissions, AA : Permissions \times Actions \}$.

Given a SecureUML+ComponentUML model $M$, one defines a $\Sigma_{RBAC}$-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 $UserAssignment$, $PermissionAssignment$, and $ActionAssignment$. Additionally, we define the partial orders $\geq_{Roles}$ and $\geq_{Actions}$ on the sets of roles and actions, respectively. $\geq_{Roles}$ is given by the reflexive-transitive closure of the metamodel association $RoleHierarchy$ on $Role$, and we write subroles (roles with additional privileges) on the left (larger) side of the $\geq$-symbol. $\geq_{Actions}$ is given by the transitive closure of the composition hierarchy on actions, defined by the metamodel association $ActionHierarchy$. We write $a_1 \geq_{Actions} a_2$, if $a_2$ is a subordinated action of $a_1$. These relations are partial orders since aggregations in UML are antisymmetric by definition.

Remark: Let $\Im_{RBAC}$ be the $\Sigma_{RBAC}$ 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 $M$.

<table>
<thead>
<tr>
<th>OCL Expression</th>
<th>$M$ Satisfies</th>
<th>$\Im_{RBAC}$ Evaluates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$UA(u, r)$</td>
<td></td>
<td>$u \cdot hasrole \to includes(r)$</td>
</tr>
<tr>
<td>$PA(r, p)$</td>
<td></td>
<td>$r \cdot haspermission \to includes(p)$</td>
</tr>
<tr>
<td>$AA(p, a)$</td>
<td></td>
<td>$p \cdot accesses \to includes(a)$</td>
</tr>
</tbody>
</table>

Next, we introduce the formulas

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

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

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

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

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 $\geq_{Roles}$ in the $\Sigma_{RBAC}$-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.supero)
role in the role hierarchy. Also, we use these operations to define the operation\texttt{Role::allPermissions()().Set(Permission)} that returns the collection of permissions (directly or indirectly) assigned to a role.

\begin{verbatim}
context Role::allPermissions():Set(Permission) body:
  self.superoPlus().haspermission -> asSet()
\end{verbatim}

In our example, \texttt{Supervisor::allPermissions()} evaluates to \texttt{Set(Permission)} on \texttt{SCHEDULER}.

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

To analyze the relation \(\geq_{\text{Actions}}\) in the \(\Sigma_{\text{RBAC}}\)-structure \(\mathcal{M}_{\text{RBAC}}\) defined by a model \(M\), we define the operation\texttt{Action::subactionPlus():Set(AtomicAction)} that returns the collection of atomic actions (directly or indirectly) subordinated to an action.

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

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

Similarly, we define \texttt{Action::compactionPlus():Set(AtomicAction)} returning the collection of actions to which an action is (directly or indirectly) subordinated. In addition, we define the operation \texttt{Permission::allRoles():Set(Permission)} that returns the collection of atomic actions whose access is (directly or indirectly) granted by a permission.

\begin{verbatim}
context Permission::allRoles():Set(Permission) body:
  self.accesses.subactionPlus() -> asSet()
\end{verbatim}

In our example, \texttt{OwnerMeeting::allRoles()} evaluates to \texttt{Set(MeetingAtomicDelete, MeetingstartAtomicUpdate, MeetingdurationAtomicUpdate, MeetingcancelAtomicUpdate, MeetingnotifyAtomicExecute, MeetingownerAtomicUpdate, MeetingparticipantsAtomicUpdate)} on \texttt{SCHEDULER}.

Conversely, we define \texttt{Action::allAssignedPermissions():Set(Permission)} returning the collection of permissions that (directly or indirectly) grant access to an action.

Finally, we define the operation \texttt{User::allAllowedActions():Set(Permission)} that returns the collection of atomic actions that are permitted for the given user, subject to the satisfaction of the associated constraints in each concrete scenario.

\begin{verbatim}
context User::allAllowedActions():Set(AtomicAction) body:
  self.hasrole.allPermissions().allActions() -> asSet()
\end{verbatim}

\textbf{Remark:} Let \(\mathcal{M}_{\text{RBAC}}\) be the \(\Sigma_{\text{RBAC}}\) structure defined by a model \(M\). Then, for any \(u\) in \texttt{Users}, \(r_1\), \(r_2\), and \(p\) in \texttt{Roles}, \(a_1\) in \texttt{Actions}, and \(a\) and \(a_2\) in \texttt{AtomicActions}, Table 1 shows the correspondence between satisfaction in \(\mathcal{M}_{\text{RBAC}}\) and evaluation of OCL expressions in \(\mathcal{M}\).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
is satisfied in \(\mathcal{M}_{\text{RBAC}}\) & evaluates to true in \(\mathcal{M}\) \\
\hline
\(r_1 \geq_{\text{Roles}} r_2\) & \(r_1\).superoPlus() -> includes\((r_2)\) \\
\hline
\(\exists r_2 \in \text{Roles}, r_1 \geq_{\text{Roles}} r_2 \land \text{PA}(r_2, p)\) & \(r_1\).allPermissions() -> includes\((p)\) \\
\hline
\(a_1 \geq_{\text{Actions}} a_2\) & \(a_1\).subactionPlus() -> includes\((a_2)\) \\
\hline
\(\phi_{\text{User}}(u, p)\) & \(u\).hasrole.allPermissions() -> includes\((p)\) \\
\hline
\(\phi_{\text{Action}}(p, a)\) & \(p\).allRoles().includes\((a)\) \\
\hline
\(\phi_{\text{RBAC}}(u, a)\) & \(u\).allAllowedActions() -> includes\((a)\) \\
\hline
\end{tabular}
\caption{Correspondence between Semantics and OCL Evaluation.}
\end{table}

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, one must be able to specify system states, which we will call \textit{security scenarios} in the following.

In a security scenario, one 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, \textit{Bob} and \textit{Alice}, with roles \texttt{SystemUser} and \texttt{Supervisor} respectively. It also contains one meeting, \textit{Kick-off}, whose owner is \textit{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. Afterwards, we must define how to map the security-design model together with the security scenario to a snapshot of this new metamodel.

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 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 ensure 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:

\[
\text{context A inv:} \\
A.allInstances() \rightarrow \forall (x, y) \ x.\text{entity} = y.\text{entity}
\]

\[
\text{context A inv:} \\
B.allInstances() \rightarrow \forall (b) \ b.\text{entity} <\rightarrow \text{self.entity}
\]

(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:

\[
\text{context Resource root().Entity post:} \\
\quad \text{if selfoclIsTypeOf(Entity) then self} \\
\quad \text{else if selfoclIsTypeOf(Method) then self.entity} \\
\quad \text{else if selfoclIsTypeOf(Attribute) then self.entity} \\
\quad \text{else if selfoclIsTypeOf(AssociationEnd)} \\
\quad \quad \text{then self.entity} \\
\quad \text{else OclUndefined} \\
\quad \text{endif endif endif endif}
\]

\[
\text{context AtomicAction} inv: \\
\quad \text{self.actionInstance} \rightarrow \text{size()} = \\
\quad \text{self.resource.root().entityInstance} \rightarrow \text{size()} \\
\quad \text{self.resource.root().entityInstance} \rightarrow \forall (e) (e \rightarrow \exists (a) \\
\quad \text{a.actionInstance} \rightarrow \exists (a) ai.resourceInstance = e))
\]

\[
\text{context ActionInstance inv:} \\
\quad \text{self.action.resource.root()} = \\
\quad \text{self.resourceInstance.entity}
\]

(iv) Finally, a method

\[
\text{AuthorizationConstraint::evaluate(caller:User, self.EntityInstance):Boolean}
\]

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. 

Figure 4. Example Scenario
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 straightforward: 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 is linked both to the object MeetingstartAtomicRead of the class AtomicAction (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 $\Sigma_{ST}$ be the order-sorted signature corresponding to the ComponentUML model and let $\mathcal{S}_{ST}$ be the $\Sigma_{ST}$-structure defined by the security scenario. For the formal definition of $\Sigma_{ST}$ and $\mathcal{S}_{ST}$, we refer to [4].

Basically, $\Sigma_{ST}$ 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 $\mathcal{S}_{ST}$ from the security scenario is the straightforward one. Let $\Sigma_{AC} = \Sigma_{RBAC} \cup \Sigma_{ST}$ and $\mathcal{S}_{AC} = (\mathcal{S}_{ST}, \Sigma_{RBAC})$.

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

$$\mathcal{S}_{AC} \models \forall p \in \text{Permissions} \phi_{User}(u,p) \land \phi_{Action}(p,a) \land \phi_{ST}^p(u) \ ,$$

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

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

$$\mathcal{S}_{ST} \models \forall p \in \text{UAP}(u,a) \phi_{ST}^p(u) \ .$$

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

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

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

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

$$\text{User::allAuthConstUser(a:Action)}: \text{Set(AuthorizationConstraint)}$$

that returns the list of authorization constraints for a given user-action pair.
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 \( M \), possibly with additional arguments referring to the objects in \( M \). 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?

```ocl
class Role { 
  allAuthConstRole(a: Action): Set[AuthorizationConstraint] body: 
  self.hasRole(superrolePlus()).allAuthConstRole(a) 
}
```

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

```ocl
class AtomicAction { 
  allAssignedRoles(): Set[Roles] body: 
  self.compactionPlus().isAssigned.allRoles() 
}
```

In our example, `SystemAdministrator.allAtomic()` evaluates to `Set{MeetingstartAtomicRead, MeetingdurationAtomicRead, MeetingownerAtomicRead, MeetingparticipantsAtomicRead, PersonnameAtomicRead, PersonmeetingAtomicRead, PersonAtomicCreate, PersonAtomicDelete}` on SCHEDULER.

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

```ocl
class Role { 
  allAuthConst(a: Action): Set[Role] body: 
  self.permissionPlus(a).isAssigned.allRoles() 
}
```

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

```ocl
class Role { 
  duplicateRoles(): Set[Role] body: 
  Role.allInstances() 
  |>exists(r1, r2 | r1.allAtomic() = r2.allAtomic()) 
}
```

6.2. User authorization

Example: Given a user, what actions are allowed to perform?

```ocl
class User { 
  allAllowedActionInstances(): Set[ActionInstance] body: 
  self.allAssignedRoles().allAssignedUsers().actionInstance > select(ai | self.isAllowed(ai)) 
}
```

Example: Given a user, what actions are allowed to perform?

```ocl
class User { 
  isAllowed(a: Action): Boolean body: 
  self.allAuthConstUser(a.action) > exists(au | au.evaluate(self, a.resourceInstance)) 
}
```

Example: Given a user, what actions are allowed to perform?

```ocl
class User { 
  allAllowedActionInstances(): Set[ActionInstance] body: 
  self.allAssignedRoles().allAssignedUsers().actionInstance > select(ai | self.isAllowed(ai)) 
}
```

In our example, `Kick-OffAtomicDelete.allUsers()` 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.

In our example, `Anonymous.allAllowedActionInstances()` evaluates to `Set{Kick-OffAtomicCreate, Kick-OffstartAtomicRead, Kick-OffdurationAtomicRead, Kick-OffownerAtomicRead, Kick-OffparticipantAtomicRead}` on SCHEDULER+.

Table 2

<table>
<thead>
<tr>
<th>is satisfied in ( \mathcal{S}_{AC} )</th>
<th>evaluates to true in ( \mathcal{M} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists p \in \text{Permissi}ns ) &amp; ( p \in \text{UAP}(u, a) ) &amp; ( \phi^p_{ST} )</td>
<td>( u.\text{allAuthConstUser}(a) \rightarrow \text{includes}(\phi^p_{ST}) )</td>
</tr>
</tbody>
</table>

Table 2

Correspondence between Semantics and OCL Evaluation.
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.

\[
\text{context AtomicAction::minimumRole(): Set(Role) body:}
\text{self.allAssignedRoles() -> select(r1 | self.allAssignedRoles() -> forAll(r2 | r1.allAtoms() -> size() <= r2.allAtoms() -> size()))}
\]

In our example, User::allRolesToPerform(Kick-OffstartAtomicUpdate) respectively to Set() and Set(Supervisor) on SCHEDULER+

Example: Are there roles to perform the action in the given scenario?

\[
\text{context User::allRolesToPerform(ai: ActionInstance): Set(Role) body:}
\text{self.allAssignedRoles() -> select(r | r.allAuthConstRole(ai.action) -> exists(a) au.evaluate(self, ai.resourceInstance))}
\]

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?

\[
\text{context ActionInstance::accessAllUsers(): Boolean body:}
\text{ActionInstance.allInstances() -> exists(ai | User.allInstances() -> forAll(u | u.isAllowed(ai)))}
\]

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, entering 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.

SecureMOVA extends MOVA to support the automated analysis of security-design models. In particular, it allows users to draw SecureUML+ComponentUML models and scenarios, to write OCL authorization constraints and assign them to permissions, and to enter 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. Second, 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.
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:

\[
\text{context User inv:}
\]

\[
\text{let M : Set = \{ \{accounts payable manager, purchasing manager\}, \ldots \} in}
\]

\[
M \rightarrow \text{select}(m | \text{self.role} \rightarrow \text{intersection}(m) \rightarrow \text{size} > 1) \rightarrow \text{isEmpty()}
\]

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.supperrolePlus() ->intersection(m)->size > 1)
->isEmpty()

From our definition of supperrolePlus() 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 and 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 questions 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
The language includes predefined types like Boolean, Integer, and String, with standard operators like +, −, *, 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} −→ collect(ii + 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.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 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 +, −, *, and / 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

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:
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

context Permission inv defaultPermissionAssignedToDefaultRole:
  self.default implies self.givesaccess ->forall(r | r.default)

context Permission 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()

context AtomicAction 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 containsSubactions:
  self.subordinatedactions = self.resource.action 
  ->select(a | a.oclIsTypeOf(EntityFullAccess)) 
  ->union(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(EntityCreate)))
context EntityRead inv containsSubactions:
  self.subordinatedactions = self.resource.oclAsType(Entity).hasattribute.action 
  ->select(a | a.oclIsTypeOf(AssociationEnd)) 
  ->union(self.resource.oclAsType(Entity) 
  .hasassociationend.action

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(AssociationEnd)) 
  ->union(self.resource.action 
  ->select(a | a.oclIsTypeOf(AssociationEnd))) 
  ->union(self.resource.action 
  ->select(a | a.oclIsTypeOf(AssociationEnd)))
For each user \(u\), for each inheritance relationship between two roles, the contextualization of the mapping from SecureUML+ComponentUML to Abstract Models.

### 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 the value true for its default-attribute.
- For each role \(r\), insert (i) an object \(\tau\) of the class Role, with the value false for its default-attribute, and (ii) a RoleHierarchy-link between \(\tau\) (subrole) and “default”.
- For each inheritance relationship between two roles \(r_1\) (subrole) and \(r_2\) (subrole), insert a RoleHierarchy-link between \(\tau_1\) and \(\tau_2\).
- For each user \(u\), insert (i) an object \(\pi\) of the class User, and (ii) a UserAssignment-link between \(\pi\) and the object “default” of the class Role.
- For each assignment of a user \(u\) to a role \(r\), insert a UserAssignment-link between \(\pi\) and \(\tau\).
- For each entity \(e\), insert (i) an object \(\Pi\) of the class Entity; (ii) an object \(efa(\tau)\) of the class EntityFullAccess; (iii) an object \(eu(\tau)\) of the class EntityUpdate; (iii) an object \(er(\tau)\) of the class EntityRead; (iii) an object \(ae(\tau)\) of the class AtomicCreate; (iv) an object \(ad(\tau)\) of the class AtomicDelete; (v) ResourceAssignment-links between \(\tau\) and \(efa(\tau)\), \(\tau\) and \(eu(\tau)\), \(\tau\) and \(er(\tau)\), \(\tau\) and \(ae(\tau)\), and \(\tau\) and \(ad(\tau)\); (vi) ActionHierarchy-links between \(efa(\tau)\) (subordinatedAction) and \(efa(\tau)\), \(er(\tau)\) (subordinatedAction) and \(efa(\tau)\), \(ae(\tau)\) (subordinatedAction) and \(efa(\tau)\), and \(ad(\tau)\) (subordinatedAction) and \(efa(\tau)\); and (vii) ActionAssignment-links between \(ae(\tau)\) and \(ad(\tau)\) and the object “default” of the class Permission.
- For each attribute \(a\) of an entity \(e\), insert (i) an object \(\Pi\) of the class Attribute; (ii) an object \(afa(\Pi)\) of the class ActionFullAccess; (iii) an object \(au(\Pi)\) of the class AtomicUpdate; (iii) an object \(ar(\Pi)\) of the class AtomicRead; (v) ResourceAssignment-links between \(\Pi\) and \(afa(\Pi)\), \(\Pi\) and \(au(\Pi)\), and \(\Pi\) and \(ar(\Pi)\); (iv) ActionHierarchy-links between \(au(\Pi)\) (subordinatedAction) and \(afa(\Pi)\), \(ar(\Pi)\) (subordinatedAction) and \(afa(\Pi)\), and \(au(\Pi)\) (subordinatedAction) and \(er(\Pi)\); (v) an EntityAttribute-link between \(\Pi\) and \(\Pi\); and (vi) ActionAssignment-links between \(au(\Pi)\) and \(ar(\Pi)\) and the object “default” of the class Permission.
- For each query method \(m\) of an entity \(e\), insert (i) an object \(\Pi\) of the class Method, with value true for its isQuery-attribute; (ii) an object \(ac(\Pi)\) of the class AtomicExecute; (iii) a ResourceAssignment-link between \(\Pi\) and \(ac(\Pi)\); (iv) an ActionHierarchy-link between \(ac(\Pi)\) (subordinatedAction) and \(er(\Pi)\); (v) an EntityMethod-link between \(\Pi\) and \(\Pi\); and (vi) an ActionAssignment-link between \(ac(\Pi)\) and the object “default” of the class Permission.
- For each association-end \(d\) of an entity \(e\), insert (i) an object \(\Pi\) of the class AssociationEnd; (ii) an object \(dfa(\Pi)\) of the class AssociationEndFullAccess; (iii) an object \(au(\Pi)\) of the class AtomicUpdate; (iii) an object \(ar(\Pi)\) of the class AtomicRead; (v) ResourceAssignment-links between \(\Pi\) and \(dfa(\Pi)\), \(\Pi\) and \(au(\Pi)\), and \(\Pi\) and \(ar(\Pi)\); (iv) ActionHierarchy-links between \(au(\Pi)\) (subordinatedAction) and \(dfa(\Pi)\), \(ar(\Pi)\) (subordinatedAction) and \(du(\Pi)\), and \(au(\Pi)\) (subordinatedAction) and \(er(\Pi)\); (v) an EntityAssociationEnd-link between \(\Pi\) and \(\Pi\); and (vi) an ActionAssignment-link between \(au(\Pi)\) and \(ar(\Pi)\) 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 \(ath\), with values “OCL” and “ath” for its language and body attributes, respectively.
- Insert an object “default” of the class Permission, with the value true for its default-attribute.
- Insert a ConstraintAssignment-link between the “default” object of the class Permission and the “default” object of the...
For each entity e and each permission p that grants “entity full access” to e, insert an ActionAssignment-link between the “default” object of the class Permission and ad(e). 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 ac(e). 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(e), ar(e), or ae(e).

For each entity e and each permission p that grants “entity read access” to e, insert an ActionAssignment-link between er(e) and 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 ae(e). 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(e), ar(e), or ae(e).

For each attribute a and each permission p that grants “attribute full access” to a, insert an ActionAssignment-link between a fa(a) and p. Delete any ActionAssignment-link between the “default” object of the class Permission and au(a) or ar(a).

For each attribute a and each permission p that grants “attribute read access” to a, insert an ActionAssignment-link between ar(a) and p. Delete any ActionAssignment-link between the “default” object of the class Permission and au(a) or ar(a).

From security-policy scenarios to abstract models.

For each object ei of an entity e, insert (i) an object ei of the class Entity; (ii) an Entity–EntityInstance link between e and ei; (iii) an object ac(ei) of the class AtomicActionInstance; (iv) an object ad(ei) of the class AtomicActionInstance; (v) EntityInstance–ActionInstance links between ei and ac(ei), and ad(ei) and ac(ei); and (vi) Action–ActionInstance links between ac(ei) and ac(ei), and ad(ei) and ad(ei).

For each attribute a of an entity e, and each object ei of this entity, insert (i) an object ar(ei) of the class ActionInstance; (ii) an object au(ei) of the class ActionInstance; (iii) Action–ActionInstance links between ar(ei) and ar(ei), and au(ei) and au(ei); and (iv) ActionInstance–EntityInstance links between ar(ei) and ac(ei), and au(ei) and ac(ei).

For each method m of an entity e and each object ei of this entity, insert (i) an object aei(ei) of the class ActionInstance; (ii) an Action–ActionInstance link between aei(ei) and aei(ei); and (iii) an ActionInstance–EntityInstance link between aei(ei) and ac(ei).

For each association-end d of an entity e, and for each object ei of this entity, insert (i) an object ar(ei) of the class ActionInstance; (ii) an object au(ei) of the class ActionInstance; (iii) Action–ActionInstance links between ar(ei) and ar(ei), and au(ei) and au(ei); and (iv) ActionInstance–EntityInstance links between ar(ei) and ac(ei), and au(ei) and ac(ei).