UML Specification of Access Control Policies and their Formal Verification

Manuel Koch[1] and Francesco Parisi-Presicce[23]

[1] Freie Universität Berlin, Berlin (DE)
[2] Univ. di Roma La Sapienza, Rome (IT)
[3] George Mason Univ., Fairfax (VA - USA)
mkoch@inf.fu-berlin.de, parisi@dsi.uniroma1.it, fparisi@ise.gmu.edu

Abstract. Security requirements have become an integral part of most modern software systems. In order to produce secure systems, it is necessary to provide software engineers with the appropriate systematic support. We propose a methodology to integrate the specification of access control policies into UML and provide a graph-based formal semantics for the UML access control specification which permits to reason about the coherence of the access control specification. The main concepts in the UML access control specification are illustrated with an example access control model for distributed object systems.

1 Introduction

Security requirements are an important aspect in the development of software systems. In order to increase the overall system security and to more easily satisfy the security constraints, security requirements should be taken into account early in the software development process. Security requirements should be considered early to avoid integration difficulties and unsatisfied security requirements. Furthermore, the specification of security requirements should be easy, to enable the software engineer to integrate security aspects into the application without being a security expert. Finally, it must be possible to describe the security requirements using models known to software engineers. Besides an easy specification, the use of known model techniques prevents software engineers from specifying mistakes in the security specifications. Once the requirements are specified, known model techniques allow their verification to ensure security properties.

Since the UML is nowadays the de-facto standard modelling language, the usage of UML for the specification of security aspects is an attractive aim, since software engineers are used to the UML notation and the accompanying tools [19, 5, 15, 7, 10]. We consider in this paper the specification of access control (AC) models with UML. In our proposal, AC models are modelled using existing UML model elements, so that UML tools can be directly used for the AC specification. Class diagrams are used to specify the AC model entities, object diagrams are used for AC policy rules, for AC model constraints and the access decision function.

We give a graph-based formal semantics to the UML AC specification by the transformation of the UML diagrams of the UML AC specification into attributed graphs and graph rules [21]. The transformation gives a graph-based security framework, for which we suggested verification concepts in [16–18].

Section 2 introduces the AC model used as the running example of this article; section 3 presents the components of a UML AC specification and section 5 is an application of the UML AC specification to a hospital application. Section 7 gives a formal semantics based on graph transformations to the UML AC specification and section 8 shows how the formal semantics can be used to prove some properties of the UML AC specification. Section 10 concludes the paper and points to future work.

2 Access Control in object-oriented systems

An access control model specifically designed to support the design and management of access control policies in object-oriented systems is *view-based access control* (VBAC) [2, 3]. VBAC relies on roles as abstractions of callers and can be regarded as an extension of *role-based access control* (RBAC) [22] to distributed object systems. Role-based access control reduces the complexity and cost of security administration in large systems because roles serve as a link between access modes for objects (e.g. read or write access if the object is a file, the print command for a printer, a method of a distributed object etc.) and subjects (users or processes that run on behalf of them). A subject can access an object if it has a role which has the required permissions for the access.

The principal new feature of VBAC is that of a *view* for the description of fine-grained access rights, which are permissions for calling operations of distributed objects. Views on objects are assigned to roles, and a subject can call an operation of an object if it has a view on the object with a permission for the operation. The subject has no access to the operation if the operation is not in a view assigned to one of the subject's roles.

Figure 1 shows a small example for the access control of `Paper` objects providing the operations `read()`, `write()`, `append()` and `find()`. The permissions `read`, `write`, `append` and `find` give the access right to call the homonymous operation. There are two views in this example: view `Reading` consists of the permissions `read` and `find`, the view `Modifying` has the permissions `write` and `append`. The view `Reading` is assigned to role `Reviewer`, view `Modifying` to the role `Author`. Therefore, any user in role `Modifying` can call any operation of paper objects. A user in role `Reviewer`, however, has read access to a paper but cannot modify it.

VBAC access policies are delivered in descriptor files and deployed together with applications in the target environments, similar to approaches like EJB [23] or the

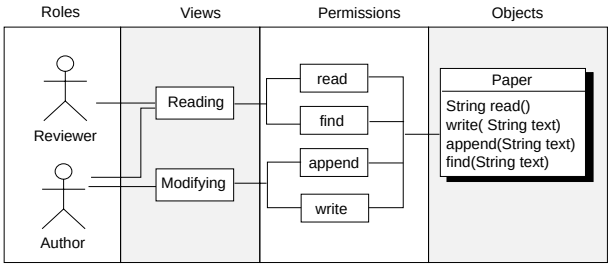

Fig. 1. VBAC example.

CORBA Component Model [20]. These policies are initially designed by developers and subsequently used and potentially adapted by deployers and security managers. The deployment and management infrastructure designed for this approach is called Raccoon [3, 4]. A deployment tool processes policy descriptors and stores static view and role definitions in repositories that can be managed using graphical management tools. At runtime, role membership is represented by digital certificates issued by a role server. Access decisions are made locally in the server processes that host application objects. These decisions are made on the basis of policy information that is supplied by policy servers, which rely on the deployed policy information. Policies can be managed in terms of roles and views using graphical management tools.

3 Access Control specification in UML

We propose a three-layer model structure consisting of an access control metamodel, an access control model and an access control instance model. We focus in this paper on the access control meta model and the access control model.

Layer	**Description**	**Example**
access control meta-model (ACMM)	Defines the access control model independent of an information/application domain	VBAC, RBAC, etc.
access control model (ACM)	Access control model regarding an information/application domain	Nurses, Doctor, etc.
instance model (IM)	instance of the model in a specific information domain/ application	surgeon, internist etc.

The access control metamodel defines the language for specifying access control models which are independent of an information or an application domain. Examples are VBAC, RBAC, Discretionary Access Control or Mandatory Access Control models.

The access control model is an instance of the access control metamodel introducing the information and application domain, respectively. If the metamodel specifies an RBAC model, doctor and nurse would be roles specific to a medical information domain.

The instance access control model is an instance of the access control model in a specific information or application domain. Examples would be orthopaedist, surgeon, internist, etc. for doctor role instance objects.

We introduce in the next section the specification of the access control metamodel and use the VBAC model as an example. Section 5 concerns the specification of the access control model. We apply the VBAC metamodel to a medical information domain.

4 Access Control Metamodel

An access control policy consists of a set of policy rules that define the choices in the individual and collective behaviour of the entities in the system. To specify an AC policy, it is necessary to model the entities in the system, the policy rules for the behaviour of these entities and (possibly) additional constraints. Therefore, a *UML access control specification* $ACS = (T, PRules, Constr)$ consists of the following UML diagrams:

- T, called *type diagram*, is a class diagram that specifies the available entities,
- $PRules$ is a set of object diagrams for the specification of the policy rules,
- $Constr$ is a set of object diagrams or OCL constraints for AC model constraints.

In the sequel, we explain each of the components using the VBAC policy model of section 2 as running example.

4.1 AC entities

The available entities in an access control model are modelled in a class diagram, called *type diagram*. For example, in the Access Control List implementation known from Unix operating systems, the entities are mainly users, groups, processes and files/directories with their read, write and execute permissions. In a Role-based access control model, major entities are the roles and their assigned permissions, users, sessions and objects.

The entities of the VBAC model are specified in the type diagram in figure 2.

Object: Represents the distributed objects in the system to which access must be controlled by a policy. Objects can be related (e.g., by inheritance).

Subject: Represents the system users or processes that run on behalf of users. Subjects may have references to objects to access them.

Permission: A permission specifies a right to access an object. The permission is uniquely assigned to the object to which the access right belongs. For one object, however, there can be several permissions.

View: A view groups a number of permissions belonging to the same object, i.e., views containing permissions defined for different objects are not allowed.

Role: Represents the access control roles which are assigned to subjects. One role can be played by several subjects and one subject can play several roles. Views are assigned to roles: one view can be assigned to different roles and a role may have several views. Roles can be related by an inheritance relation, where the extended role inherits all the views of the base view.

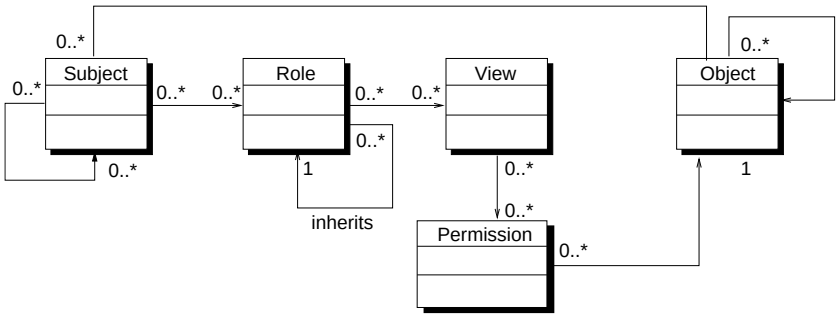

Fig. 2. *The VBAC model.*

In a concrete application, the classes in the type diagram are specialised to classes taken from the application class diagram. For example, the role class in the VBAC type diagram must be specialised to concrete roles of the given application (e.g., to the roles `Author` or `Reviewer` of fig. 1). Analog, all other classes are refined to application specific classes.

4.2 Access Control Policy Rules

The access control policy rules define the behaviour of the system entities and control the possible system states. Policy rules are specified in object diagrams using

the special stereotypes `<<create>>` and `<<destroy>>`. The intended meaning of an object or link with a stereotype `<<create>>` is that the object or link is created by the system. The intended meaning of an object or link with stereotype `<<destroy>>` is that the object or link is removed. Our approach in representing the actions of a policy rule is similar to the representation of postconditions for actions in the Catalysis approach [8]. To visualise an action in Catalysis, one diagram is used to present the state before and after the action. Catalysis, however, uses a presentation of the "after" state by bold elements or different colors instead of stereotypes.

Figure 3 (role management) and figure 4 (view management) show the policy rules for the VBAC model.

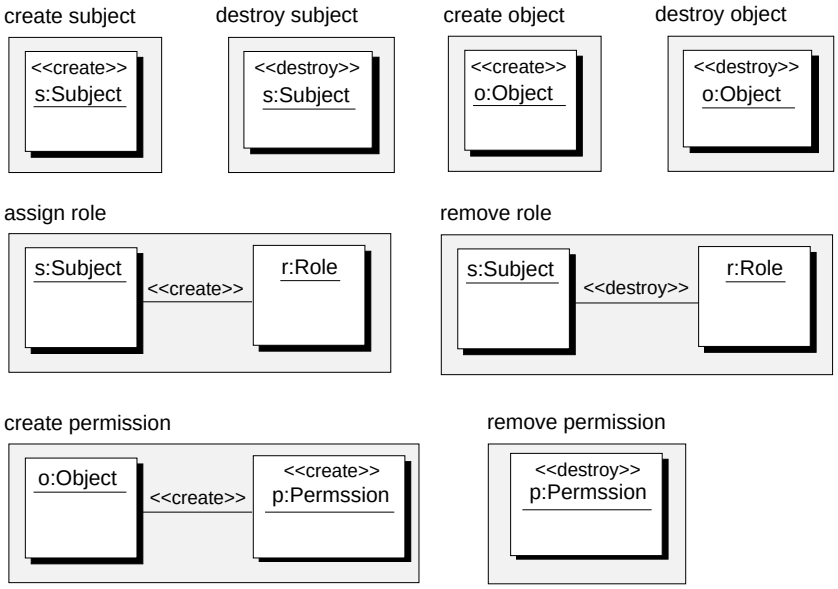

Fig. 3. *VBAC policy rules.*

create subject: Subject instances can be unconditionally added to the system at any time, that is, their creation does not depend on the current system state. The object diagram for policy rule `create subject` consists of one subject instance with the stereotype `<<create>>` which specifies that the subject instance is created by the rule.

remove subject: Subjects can be unconditionally removed from the system at any time. The object diagram for rule `remove subject` specifies this by a subject with stereotype `<<destroy>>`.

create object: Object instances can be created analog to subjects.

destroy object: Object instances can be removed from the system analog to subjects.

assign role: Rule `assign role` specifies the assignment of a subject to a role. The assignment is modelled by a link between the subject instance and the role instance. The link is created (carries stereotype `<<create>>`) while the subject as well as the role must already exist in the system (they do not have a `<<create>>`).

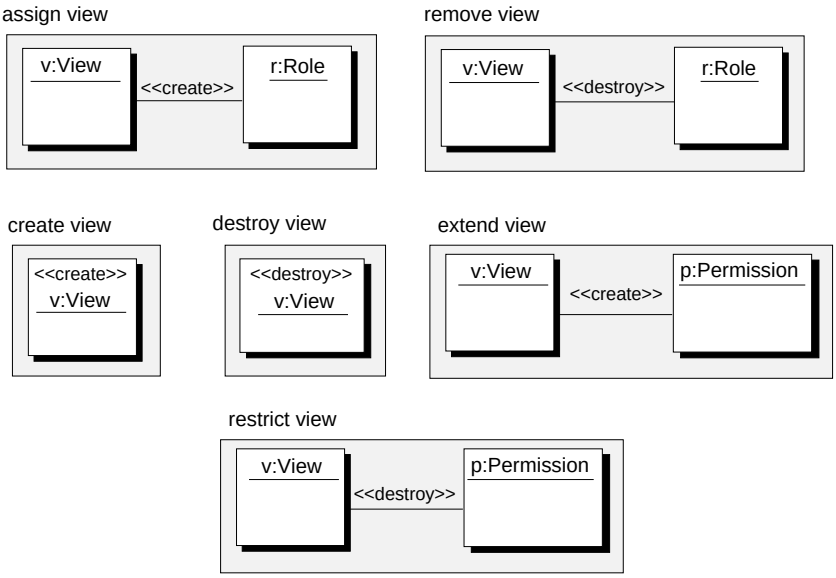

Fig. 4. *VBAC policy rules for view management.*

remove role: Rule `remove role` specifies the removal of a subject from a role by deletion of the link between the role and the subject. Both the subject and the role instance remain in the system, since they do not carry a `<<destroy>>` stereotype.

create permission: Rule `create permission` creates a new permission object. Since a permission must belong always to a unique object, the created permission object is immediately connected to an object.

remove perm.: Rule `remove permission` specifies the removal of a permission.

assign view: The assignment of a view instance to a role instance is specified by a link between the two instances. Rule `assign view` specifies the role-view assignment by the creation of a link between an existing role and an existing view.

remove view: Rule `remove view` specifies the removal of a view from a role by destroying the link between the role and the view.

create view: View instances can be created analogously to subjects.

destroy view: View instances can be removed analogously to subjects.

extend view: A view consists of a set of permissions. The assignment of a permission to a view instance is specified by a link between the instances. The rule `extend view` inserts this link between the view and the permission instance.

restrict view: The rule `restrict view` removes a permission from a view by removing the connecting link.

4.3 AC model constraints

The access control policy rules determine the acceptable system states. In some situations, however, it is necessary to reduce the system states accepted by the policy rules. For example, the policy rule `assign role` of the VBAC model can be used to assign any number of subjects to the same role (by repeated application to the same role). In many applications, however, there are role cardinality constraints, e.g. a maximal number of subjects in a role. An additional constraint in VBAC is the requirement that a view must have at least one permission when the view is assigned to a role, otherwise permissionless views are allowed.

Some cardinality constraints are already included in the access control type diagram of a UML AC specification (e.g., figure 2 requires a unique object for each permission). The VBAC example constraint which requires at least one permission for a view assigned to a role, however, is not expressible by cardinalities in a UML class diagram and must be specified separately. We present two ways to specify these additional access control constraints: textually by the Object Constraint Language (OCL) or graphically by object diagrams.

The OCL constraint for the requirement that each view has a permission when assigned to a role is as follows:

```
context View inv
   (self.role->notEmpty) implies (self.permission->notEmpty)
```

The OCL is a powerful specification technique in which complex constraints can be expressed. For a restricted set of OCL constraints we can give a graphical representation which adapts itself better to the graphical nature of the other UML

diagrams. Furthermore, in section 7 this graphical representation can then be used to apply verification concepts. The OCL constraints that can be given a graphical representation are mainly OCL constraints concerning the object structure of a system state. Some other OCL constraints can be presented graphical as well (see [1]), but this is not topic in this paper.

The graphical access control constraints are either object diagrams which show a forbidden structure that must never occur in a system state or they are object diagrams which require a structure under certain system state conditions. We call the former object diagrams *negative constraints*, the latter *positive constraints*. The object diagrams for positive constraints make use of the stereotype `<<exists>>`: all objects and links labelled with this stereotype must exist in a system state when the remaining, i.e., non `<<exists>>`-labelled, objects and links occur in a system state.

Figure 5 is the graphical specification of the view-permission requirement by a positive constraint. It shows a permission p assigned to a view v which in turn is

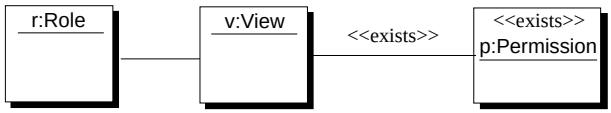

Fig. 5. *VBAC constraint*

assigned to a role r. The permission object p and its link to the view v carry the `<<exists>>` label. Both must exist when the view v is assigned to a role r. Is no role assigned to the view, the assigned permission do not have to exist. An example of a negative constraint is given in the next section.

5 Access Control Model for an Information Domain

The access control metamodel defines a language for specifying access control models. In the previous section, we introduced the access control metamodel for VBAC. An access control metamodel consists of three parts: a metamodel for the access control entities, the access control rules and the access control constraints. The access control metamodel is independent of an information domain, e.g., the VBAC meta objects are Subject, Role, View, Permission and Object. In an information domain, these metaobjects are mapped to application dependent objects.

5.1 Information Domain Access Control Entities

The access control entities in the metamodel are specified in a class diagram. The access control entities in the access control model are specified in a class diagram

as well, which is an instance of the metamodel class diagram and which contains information resp. application domain specific access control entities.

Figure 6 shows an instance of the VBAC metamodel applied to a medical information domain. We consider a small part of a hospital in which patients are

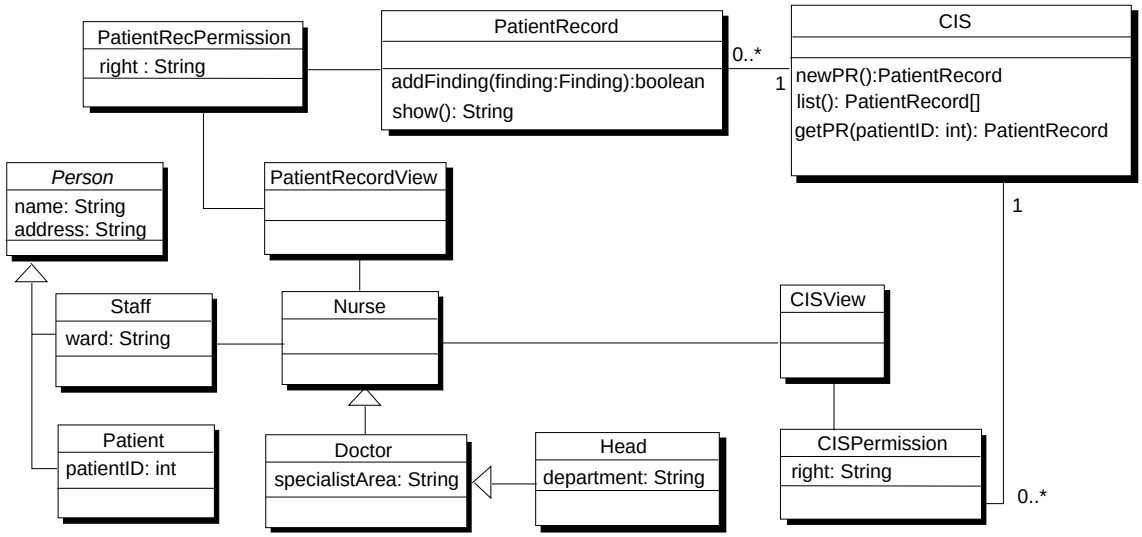

Fig. 6. *Class diagram for the application.*

medicated by doctors resp. nurses. Each ward has a head of department. The following table shows the mapping of the metaobjects to the objects in the access control model with a brief description of its meaning. We try to keep the hospital application small, but detailed enough to demonstrate the specification of the VBAC model. A more detailed introduction to the hospital application can be found in [6].

metamodel	model	description
Subject	Person	A person has a name and an address. The class is abstract.
	Staff	Specialisation of `Person`. Specifies the staff in the hospital. The attribute `ward` specifies on which ward the staff member is currently working.
	Patient	Specialisation of `Person`. Patients are identified by a patient identifier `patientID`.
Role	Nurse	Role for nurses.
	Doctor	A specialisation of role `nurse`. Each permission granted to role `nurse` is granted to doctors, too. Specialist area, e.g. surgery or orthopaedy, defines the doctor's specialist area.
	Head	Specialisation of role `Doctor`. Represents the head of department of a ward.
View	CISView	View for CIS permissions.
	Patient-RecordView	Views for PatientRecord permissions.
Objects	CIS	The central information system (CIS) contains the electronical patient records. Patient records can be created by operation `newPR()` if it is the patient's first stay in the hospital. A list of all patient records can be requested by `listPR()`, a specific patient record is returned by `getPR()`.
	Patient-Record	The (textual) contents of a patient record can be requested by operation `show()`. Whenever findings for a patient are created due to new investigations, the findings are added to the patient record by operation `addFinding()`.
Permission	CISPermission	Permissions for object `CIS`. The attribute `right` contains the name of the operation, to which the permissions grants access.
	PatientRec-Permission	Permissions for object `PatientRecord`.

5.2 Access Control Policy Rules and Constraints

The access control policy rules and the access control constraints of the access control model are derived from the object diagrams for the policy rules and constraints of the metamodel, in which metaobjects are instantiated with information domain specific

objects specified in the type diagram of the access control model. The object diagram got by instantiation of a policy rule or a constraint must be an instance of the type diagram of the access control model.

For example, the rule `assign role` in fig. 3 can be specialised to a rule in which the metaobject `s:Subject` is instantiated by the object `s:Staff` and the metaobject `r:Role` by `d:Doctor`. Another example is the meta-rule `create permission`. The metaobject `o:Object` can be instantiated by a `CIS` object, the permission object `p:Permission` by a `CISPermission` with a value `list` for attribute `right`. Fig. 7) shows this policy rule and some more examples.

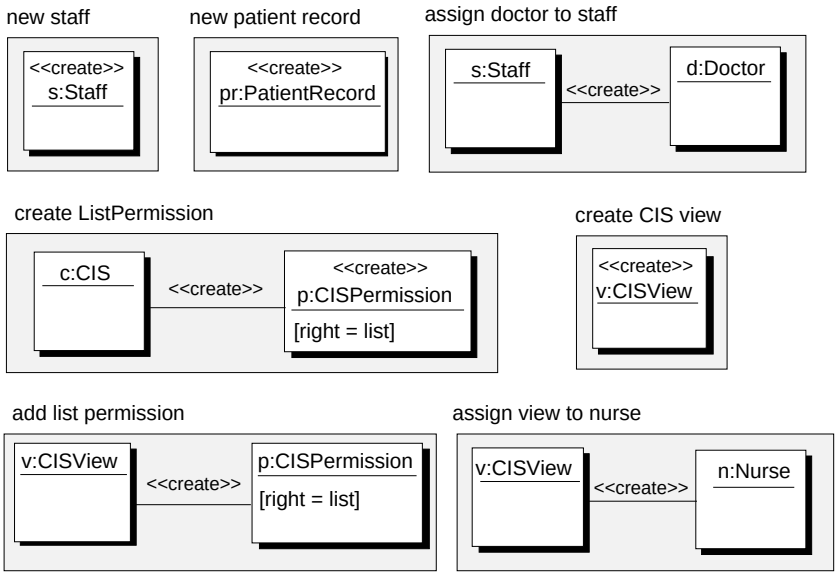

Fig. 7. *Policy rules in the access control model.*

The following instantiations are examples of invalid instantiations: The instantiation of the metaobject `s:Subject` to `p:Person` is not possible since the class `Person` is abstract, i.e., it has no instances. The instantiation of the metaobject `s:Subject` to `p:Patient` and the metaobject `r:Role` to `d:Doctor` in rule `assign role` is not possible since the type diagram of the access control model does not permit links between patient and doctor objects.

The policy rules of the access control model are given by all object diagrams that can be derived from the policy rules of the metamodel with objects taken from the access control model type diagram which are instances of the type diagram.

The access control constraints of the access control model are derived from the access control constraints of the metamodel analog to the policy rules. The meta-objects are instantiated with objects given in the access control model type diagram. An example of the specialisation of the VBAC constraint in figure 5 is a constraint in which the role is a `Nurse` object, the view is `CISView` and the permission is a `CISPermissiom`. An invalid instantiation according to the ACM class diagram is a `CISView` object for the view and a `PatientRecPermission` object for the permission.

In addition to the instantiations of metamodel constraints, there may be information domain dependent constraints. For example, the VBAC type diagram in figure 2 does not specify a restriction on the number of subjects in one role. In the hospital application, however, there is at most one head of department for each ward. The application specific constraints can be added by specifying them either graphically or with OCL. Figure 8 shows the graphical constraints for the head of department requirement. The negative constraint a) ensures that there are no more than two staff members assigned to the same `Head` role. Negative constraint b) models that there is at most one `Head` role object for each department. Below the corresponding OCL constraints are given.

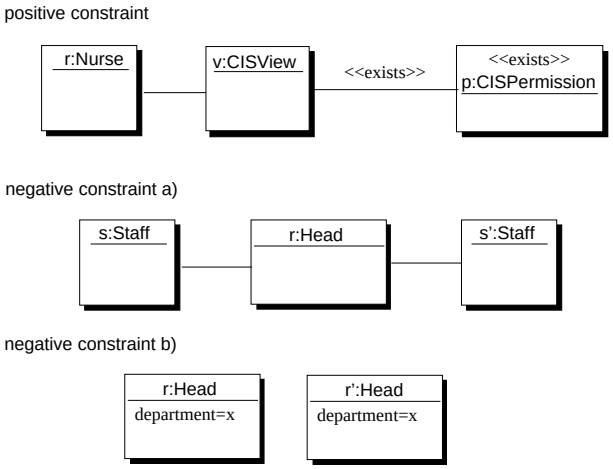

Fig. 8. *ACM constraints.*

```
a) context Head inv
      self.staff->size() < 2
b) context Head inv
      Head.allInstances->forAll(h1,h2 | h1 <> h2 implies
                                 h1.department<> h2.department )
```

The policy constraints of the access control model are given by

- all object diagrams got by the instantiation of the constraint diagrams of the access control metamodel by objects specified in the access control model (ACM) type diagram, so that the resulting object diagrams are instances of the ACM type diagram and
- a set of object diagrams specifying information domain specific constraints.

6 Access Decision Function

The access decision function of an access control model determines whether an access for a concrete subject in a particular state is possible or not. When a subject tries to access an object during runtime, the policy enforcement engine must be queried to decide whether the access is allowed or denied based on the deployed policy. In the case of the VBAC model, access control is required when object operations are called. The query comprises the question whether the subject has a role with a permission to call the desired operation.

Queries are specified by object diagrams, called *query diagrams*, which model the subject, the desired access of the subject on an object and the system structure necessary to grant this access to the subject. These three aspects may be differently modelled for different access control models, but we require them to be covered by the query diagram.

We consider access decision on the instance model layer, since the access decision enforcement engine is used during runtime when specific instances are considered. Assume for example, a runtime object `s` of type `Staff` wants to call the operation `list()` of an instance `c` of type `CIS`. The query diagram in figure 9 shows the way how the access decision function can be modelled in the VBAC model. The query diagram

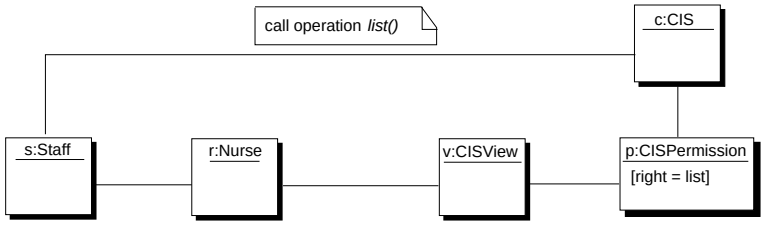

Fig. 9. *Query diagram for calling operation* `op()`.

contains the subject instance `s:Staff` which wants to call the operation `list()` of the `CIS` object `c`. The desired access of the subject (the staffer) is modelled in a note containing the operation the subject tries to call. The object structure necessary to

call the operation `list()` is that of a role assigned to the subject where the role has a view with permission `list`. In figure 9 is checked if the instance `s:Staff` is in role `Nurse` with a view having the permission `list`. The other roles must be checked as well, so that there are two additional query diagrams for role `Doctor` and `Head` as well.

To decide if a subject has access to an object in a given system configuration, the query diagram must be found in the object diagram representing the current system state. If the query diagram is a part of the system diagram, access is granted, otherwise access is denied.

7 Semantics via graph transformations

In this section, we give a graph-based, formal semantics to the UML AC specification. The formal semantics enables to check the UML AC model w.r.t. consistency properties (see section 8). We briefly introduce graph transformations in the next subsection as far as necessary for the remainder of this article. For the general concepts of graphs, graph rules, types and attribution in the algebraic approaches, see [21]. After the introduction to graph transformations, we present the translation from the UML AC model to a graph-based security policy framework [16].

7.1 Background on Graph Transformations

A *graph* consists of disjoint sets of nodes and directed edges $e : a \to b$ from a *source* node a to a *target* node b. Nodes and edges of a graph are labelled statically as well as dynamically. The static labels are node and edge types used to identify graphical objects, the dynamic labels, called *attributes*, are used to store data together with the static objects. The node and edge type information is specified in a *type graph*. For the description of attributes a simple algebraic approach is used in this article. If more complex attributes are necessary, a categorical or algebraic approach can be chosen, as well. Each node and edge type is associated with an attribute tuple type containing several attribute declarations. A declaration consists of an attribute name and an attribute data type. A graph object is associated with a tuple of constant attribute values. In rules, also variables and complex expressions are allowed.

Figure 10 shows an example of a type graph and a possible instance graph. There is one node type *class* and two edge types *generalisation* and *association*. Attributes for *class* nodes are *name* (of type `string`), stereotype (a set of `Strings`), *attributes* (set of pairs (attribute name, attribute type)) and *operations* (set of triples (operation name, operation parameter, return type). Attribute for the *association* edges is *name*, *generalisation* edges do not have attributes. Graph b) shows an instance graph with respect to the type graph.

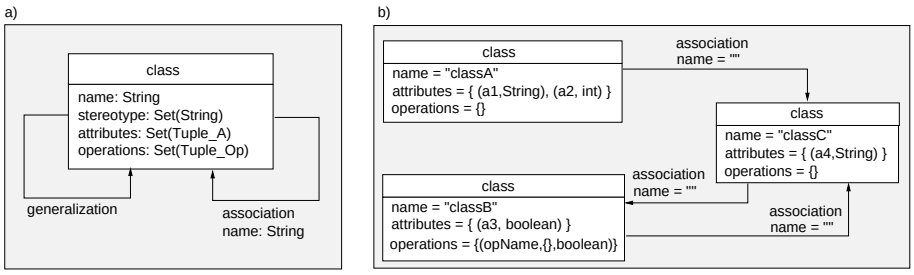

Fig. 10. *Type graph (a) and instance graph (b).*

A *graph morphism* $f : G \to H$ between two graphs G and H is a partial mapping between the nodes (resp. edges) of G and the nodes (resp. edges) of H so that 1) f respects the graph structure, i.e. whenever the mapping for edges is defined for an edge e, the mapping for the source node s and the target node t of e is defined and $f(s)$ and $f(t)$ are the source and target node for the edge $f(e)$ in H, 2) nodes and edges are mapped only to nodes and edges of the same type, and 3) f respects the attribution, i.e., attributes coincide or are in a previously defined relation (what is defined in the algebraic specification of the attribute type). We call the graph morphism *total* if the mappings between the node and edge sets are total. A graph morphism is *injective* if the underlying mappings for nodes and edges are injective. Figure 11 shows an example of an injective and total graph morphism. Please note, that the attributions of the association edges does not coincide. In graph G the name is a variable x and in graph H it is the constant "". We assume in the rest of the paper a relation between variables and constants, so that we can substitute variables by constants.

A *graph rule* r, or just *rule*, is an injective graph morphism $r : L \to R$ in which the attributes of image and domain nodes/edges coincide. The graph L, called the *left-hand side* of the graph rule, describes the elements a graph must contain for r to be applicable. The partial morphism is undefined on nodes/edges that are intended to be deleted, defined on nodes/edges that are intended to be preserved. Nodes and edges of R, *right-hand side*, without a pre-image are newly created. Note that the actual deletions/additions are performed on the graphs to which the rule is applied. An example graph rule is shown on top of figure 12. Its left-hand side consists of one class A and an association. The right-hand side has a *class* A node and a *class* C node. Both nodes are connected by an association edge. The graph morphism for the rule is defined only for the *class* A node and undefined on the association edge. Therefore, the node will be preserved during the rule application, the association edge will be deleted. Since the *class* C node and the association edge do not have a pre-image in the left-hand side, they are created by the rule.

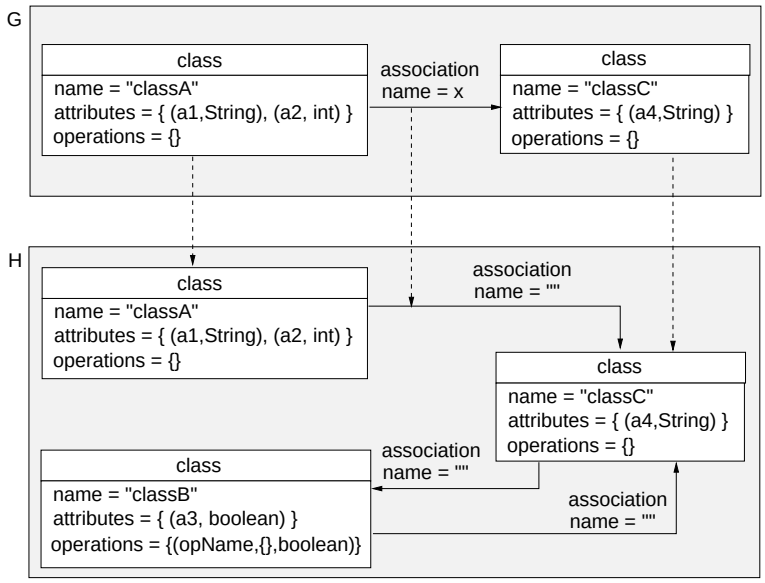

Fig. 11. *Example of a graph morphism.*

The application of a rule $r : L \to R$ to a graph G requires a total morphism $m : L \to G$, called *match*, from the left-hand side of the rule to G. The application itself is formally characterised by the pushout of the rule morphism and the match in the category of graphs and graph morphisms. The construction of the derived graph consists of two steps: first delete all objects in G that have a pre-image in $L \setminus dom(r)$, then add all graph objects of $R \setminus r(L)$ to G connected to the nodes $m(dom(r))$. If the label or attribute value of the left-hand side is a variable, it is substituted with the label or attribute value of the mapped graph object. Figure 12 applies the rule on the top to graph H. First, the association edge at *class A* in G is deleted, then the *class C* together with the association edge is added to G. The result graph is graph H.

7.2 From UML Diagrams to Graphs

Due to the graphical notation of UML, UML diagrams can be represented as attributed typed graphs. We follow the representation introduced in [25].

Each class in a class diagram is a node of type *class*. The class name is stored in the attribute *name*, the stereotype(s) in the attribute *stereotype*. The UML class attributes and operations are represented as sets of tuples in the attributes *attributes* and *operations*, respectively. Each tuple for a UML class attribute contains information about the attribute's name and the attribute type. The tuple for the UML

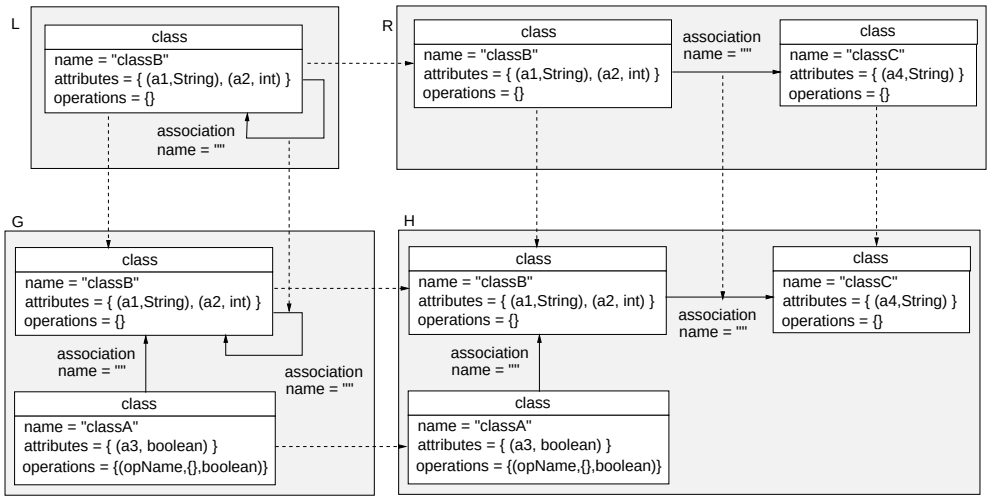

Fig. 12. *Graph rule application.*

class operations has tuple elements for the operation's name, the parameter list and the return type. The parameter list itself is a set of pairs consisting of parameter name and parameter type. Fig. 13 shows the graph representation of the UML class `PatientRecord` in fig. 6.

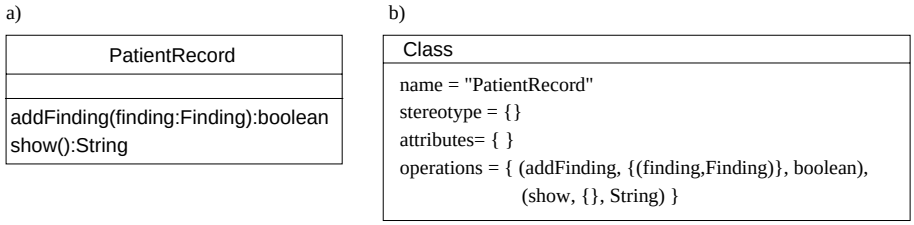

Fig. 13. *Representation of a UML class (left) as an attributed graph node (right).*

UML objects are represented in a similar way. The graph node for an UML object has the graph attributes *name* for the object name, *class* for the class name of which the object is an instance of and *attribute* for the attribute values.

Directed associations of the UML class diagram are represented as edges of type *association*. In this paper, the edge attributes contain only the association name, but other information about role names, role visibilities etc. can be transformed into edge attributes, too. Bidirectional associations are translated into two edges in opposite directions. Generalisations are represented by an edge of type *generalisation*. Each

subclass node inherits the *association* edges of the superclass node and the attributes *attribute* and *operations* of the superclass node are added to the attributes and operations of the subclass node. Fig. 14 shows an example. Links between objects

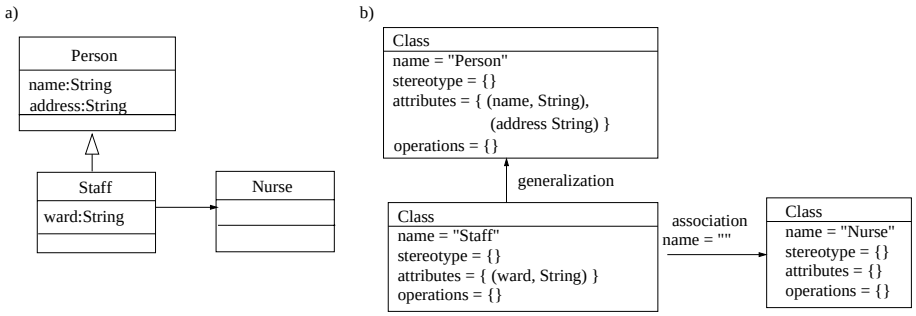

Fig. 14. *Representation of UML associations and generalisations.*

are represented by edges of type *link*.

The multiplicity information in class diagrams is not yet considered in the translation. Each multiplicity range $n..m$ $(n \leq m)$ for an association is translated into a *positive graphical constraint* for the lower bound n and a *negative graphical constraint* for the upper bound m. A positive graphical constraint is a total and injective graph morphism $c : X \rightarrow Y$ and a graph G *satisfies* c if for all total and injective graph morphisms $p : X \rightarrow G$ there exists a total and injective graph morphism $q : Y \rightarrow G$ so that $X \stackrel{c}{\rightarrow} Y \stackrel{q}{\rightarrow} G = X \stackrel{p}{\rightarrow} G$. A negative graphical constraint is a graph C and a graph G satisfies C if there does not exist a total, injective graph morphism $p : C \rightarrow G$.

The graph X of the positive graphical constraint for the lower bound n contains one object node of the association's source class. The graph Y contains the same node and n object nodes for the association's target class. The graph morphism maps the object node in X to its counterpart in Y. The graph C of the negative graphical constraint for the upper bound m contains one object node of the association's source class and $m + 1$ objects of the association's target class. Fig. 15 shows an example for a multiplicity 1..1 between class `PatientRecord` and class `CIS`. References are represented by *link* edges. A lower bound 0 and an upper bound $*$ are not translated into graphical constraints, because they have no implementation consequences.

7.3 UML AC Specification and Security Policy Framework

The translation of UML class and object diagrams into attributed graphs is the basis for the translation of a UML AC specification $ACS = (T, PRules, Constr)$ into a graph-based security policy framework [16].

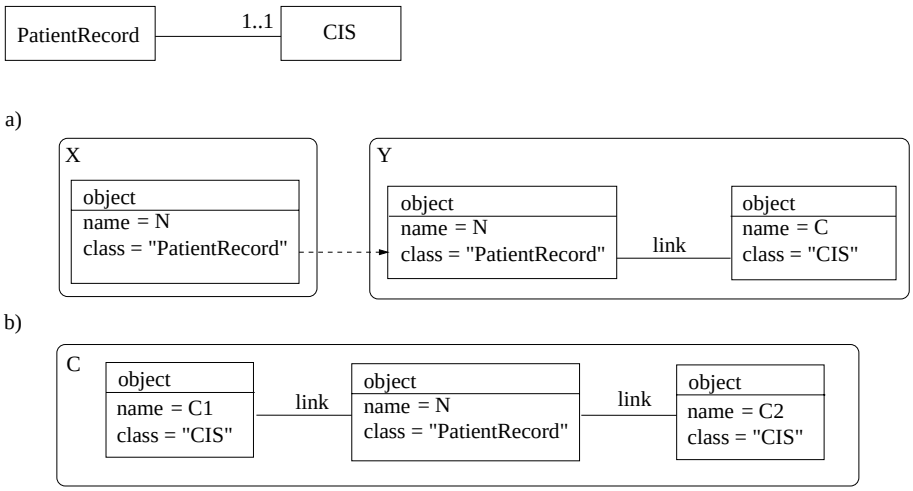

Fig. 15. *Representation of UML multiplicities in UML class diagrams: a) positive constraint for lower bound, b) negative constraint for upper bound.*

Definition 1 (Graph-based Security Policy Framework). *A* graph-based security policy framework*, or short* security framework*, is a tuple* $SP = (TG, Rules, Pos, Neg)$*, where* TG *is a type graph,* $Rules$ *is a set of graph rules,* Pos *is a set of positive graphical constraints, and* Neg *is a set of negative graphical constraints.*

Given a UML AC specification $ACS = (T, PRules, Constr)$, the type diagram T generates the type graph TG together with some positive and negative graphical constraints for the sets Pos resp. Neg, the set of object diagrams in $PRules$ generates the graph rules in $Rules$ and the object diagrams in $Constr$ generate additional positive and negative graphical constraints in Pos and Neg, respectively:

- The type diagram T of a UML AC specification is a class diagram and we take its graph representation as the type graph TG of the security framework. Since a type graph can not specify the multiplicities in a class diagram, we add the constraints for the multiplicities in the class diagram T to the graphical constraint sets Pos and Neg, respectively.
- Each object diagram OD in $PRules$ is translated into a graph rule $r : L \to R$, where L is the attributed graph given by all objects and links of OD without a stereotype `<<create>>` and R is the attributed graph given by all objects and links of OD without a stereotype `<<destroy>>`. The morphism r is defined for all objects/links without a stereotype `<<destroy>>` and maps the objects and associations to their counterparts in R. Fig. 16 shows the graph rules for the object diagrams `create subject` and `assign role` of fig. 3.

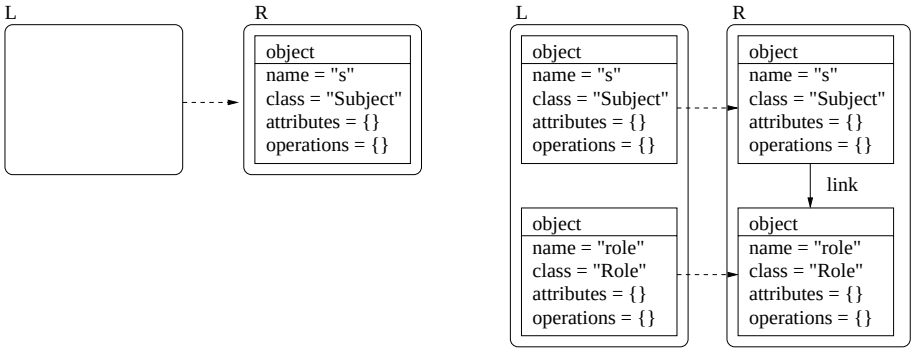

Fig. 16. *Translation of UML AC specification policy rules into graph rules.*

– Each object diagram OD in $Constr$ with an object or link carrying a stereotype `<<exists>>` is translated into a positive graphical constraint $c : X \to Y$, where X is induced by all objects/links without a stereotype `<<exists>>` and Y is the attributed graph induced by OD. The morphism c maps each node/edge to its counterpart in Y. Each object diagram OD in $Constr$ without the stereotype `<<exists>>` (i.e., a negative constraint) is translated into a negative graphical constraint C, where C is the attributed graph induced by OD. Figure 15 is the positive graphical constraint for the object diagram in figure 5.

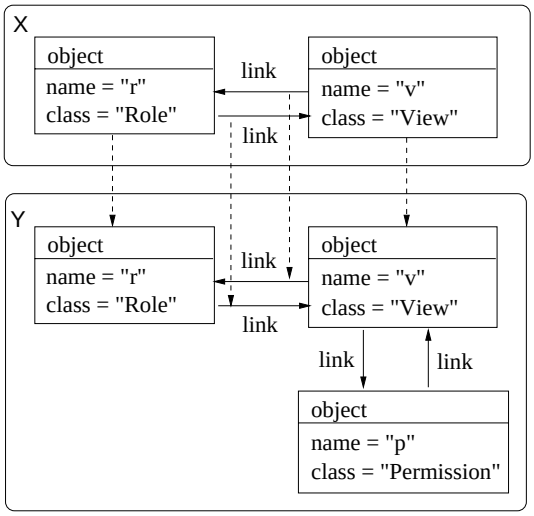

Fig. 17. *Translation of UML AC specification constraints into graphical constraints.*

8 Verification

This section concerns the *coherence* of a UML AC specification. Informally, a UML AC specification is coherent if the policy rules allow system states only which satisfy the constraints of the UML AC specification. To ensure coherence formally, we use the verification concepts in [16–18] to check coherence of a graph-based security framework. We define a UML AC specification to be coherent if the graph-based security framework is coherent which results from the translation of the UML AC specification as introduced in the previous section. A graph-based security framework is coherent if all graphs constructed by the graph rules satisfy all graphical constraints. A graph-based security framework is incoherent if there is a graph rule which creates a graph that does not satisfy one of the graphical constraints.

Consider the UML AC specification for the access control model in section 5. Its set of policy rules contains the object diagrams in figure 3 and figure 4 specialised to application specific objects given in the access control model type diagram in figure 6. One example is the specialisation of the policy rule *assign role* in figure 3 by a `Staff` instance as subject and `Head` as role. Applying this rule several times to the same `Head` role but different staff member instances, we get several staffers who play the role `Head` for the same department. The negative constraint a) in figure 8, however, forbids more than one subject in role `Head`. Therefore, the AC UML specification is not coherent.

Whether a graph rule may produce a system state which violates a graphical constraint can be detected statically [17]. In the sequel we distinguish between *deleting rules* which delete elements but do not add anything and *extending rules* which add elements but do not delete anything. All rules in this article can be put in one of these two categories.

An extending graph rule may violate a negative constraint by adding elements which complete the forbidden system state. A deleting rule cannot violate a negative constraint. An extending graph rule may violate a positive constraint $c : X \to Y$ by adding elements which construct the premise X of the positive constraint without completely constructing the conclusion Y. A deleting rule can violate a positive constraint by deleting elements from the conclusion but the premise is still valid. In the example mentioned above, the extending rule *assign role* can construct the forbidden graph of the negative constraint a) in figure 8. If a rule can violate a constraint, we say the rule and the constraint are in *conflict*.

Conflicts between rules p and constraints $c : X \to Y$ can be resolved by adding *application conditions* to the graph rules. The idea of the conflict resolution by an additional application condition is, that the application condition prevents the application of the rule whenever the application would produce a graph that violates the constraint. An application condition for a graph rule p with rule morphism

$r : L \to R$ is a total injective morphism $n : L \to N$, where the part $N \setminus n(L)$ represents a structure that must not occur in a graph G for the rule to be applicable.

Definition 2 (application condition). *An* application condition *for a rule $p : L \overset{r}{\to} R$ is a total injective graph morphism $n : L \to N$. A morphism $m : L \to G$ satisfies an application condition n if there does not exist a total injective graph morphism $q : N \to G$ with $m = q \circ n$. A rule p with application condition n is applicable to a graph G if there is a match $m : L \to G$ which satisfies the application condition.*

We present next the constructions of the application conditions that ensure a coherent graph-based security framework. We start with the construction for extending rules. Afterwards, we present the construction for deleting rules.

Definition 3 (extending rule reduction). *Given an extending rule $p : L \overset{r}{\to} R$, a graph X and a nonempty overlap S of R and X, so that $X \cap (R \setminus r(L)) \neq \emptyset$.*

$$\begin{array}{ccccc} L & \xrightarrow{r} & R & \xleftarrow{s_1} & S \\ {\scriptstyle n}\downarrow & & \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle s_2} \\ N & \xrightarrow[r^*]{} & C & \xleftarrow[b]{} & X \end{array}$$

Let $C = R +_S X$ be the pushout object of $s_1 : S \to R$ and $s_2 : S \to X$ in the category **Graph**, *and let $C \overset{r^{-1},h}{\Rightarrow} N$ be the derivation with the inverse rule $p^{-1} : R \overset{r^{-1}}{\to} L$ at match h. Define $A(p, X) = \{n : L \to N | C \overset{(r^{-1},h)}{\Rightarrow} N,\ C = R +_S X$ for some overlap S with $X \cap R \setminus r(L) \neq \emptyset\ \}$.*

Proposition 1 (preservation of satisfaction). *Let $p : L \overset{r}{\to} R$ be an extending rule, NC a negative graphical constraint and $c : X \to Y$ a positive graphical constraint.*

1. *Let $p(NC)$ be the rule p extended by the application conditions $A(p, NC)$ and $G \overset{p(c),m}{\Longrightarrow} H$ be a derivation with $p(NC)$ at match m. Then, G satisfies NC implies H satisfies NC.*
2. *Let $p(c)$ be the rule p extended by the application conditions $A(p, X)$ and $G \overset{p(c),m}{\Longrightarrow} H$ be a derivation with $p(c)$ at match m. Then, G satisfies c implies H satsifies c.*

In the case of a negative constraint, the application conditions prevent the rule from building the graph specified in the negative constraint. In the case of a positive constraint, the application conditions prevent the rule from constructing the premise X of a positive constraint. The proof is given in [12].

Next, we present the construction of the application conditions for a deleting rule.

Definition 4 (deleting rule reduction). *Given a deleting rule $p : L \xrightarrow{r} R$, a positive constraint $c : X \to Y$ and a nonempty overlap S of L and Y, so that $(L \setminus dom(r)) \cap (Y \setminus c(X)) \neq \emptyset$.*

$$\begin{array}{ccc} L & \xleftarrow{s_1} & S \\ {\scriptstyle n}\downarrow & & \downarrow{\scriptstyle s_2} \\ N & \xleftarrow[b]{} & Y \end{array}$$

Let $N = L +_S Y$ be the pushout object of $s_1 : S \to L$ and $s_2 : S \to Y$ in the category **Graph**. *Define $A(p, c) = \{n : L \to N | N = L +_S Y$ for some overlap S with $(L \setminus dom(r)) \cap (Y \setminus c(X)) \neq \emptyset\}$.*

Proposition 2 (preservation of satisfaction). *Let $p : L \xrightarrow{r} R$ be a deleting rule, $c : X \to Y$ a positive constraint and $p(c)$ the rule p extended by the application conditions $A(p, c)$ (see Constr. 4). Let $G \overset{p(c),m}{\Longrightarrow} H$ be a derivation with $p(c)$ at match m, then G satisfies c implies H satisfies c.*

Proof. Let $G \overset{p(c),m}{\Longrightarrow} H$ be a derivation and G satisfies c. Let $p : X \to H$ then there is a total morphism $p_G : X \to G$ with $r^* \circ p_G = p$ since $p(c)$ is a deleting rule. By assumption, G satisfies c so that there exists a total morphism $q_G : Y \to G$ with $c \circ q_G = p_G$. It remains to show that $r^* \circ q_G$ is total since then $r^* \circ q_G \circ c = r^* \circ p_G = p$.

We assume $r^* \circ q_G$ to be partial. Then, the rule $p(c)$ deletes parts of $Y \setminus c(X)$. By construction, there is an overlap S with morphisms $s_1 : S \to L$ and $s_2 : S \to Y$ so that $m \circ s_1 = q_G \circ s_2$. Let $(N, a : L \to N, b : Y \to N)$ be the pushout of s_1 and s_2, by pushout property, there is a unique total graph morphism $u : N \to G$ with $m = u \circ a$. This, however, is a contradiction to the fact that m is a match for $p(c)$.

For deleting rules, the application conditions prevent the rule from deleting parts of the conclusion Y which are not part of X. If parts of Y are deleted which are part of X, too, this cannot destroy satisfaction since then the premise does not exist anymore in the graph.

Theorem 1. *Let $SF = (TG, Rules, Pos, Neg)$ be a graph-based security framework and $SF' = (TG, Rules', Pos, Neg)$ the framework in which each extending rule $p \in Rules$ is extended by application conditions $A(p, NC) \cup A(p, X)$ for all $NC \in Neg$ and $c : X \to Y \in Pos$ and each deleting rule is extended by application conditions $A(p, c)$ for all $c \in Pos$. Then, SF' is coherent.*

Follows from proposition 1 and proposition 2.

An example for construction 3 is given in figure 18, an example for construction 4 in figure 19. The first example shows the extending rule *assign role* for staff members

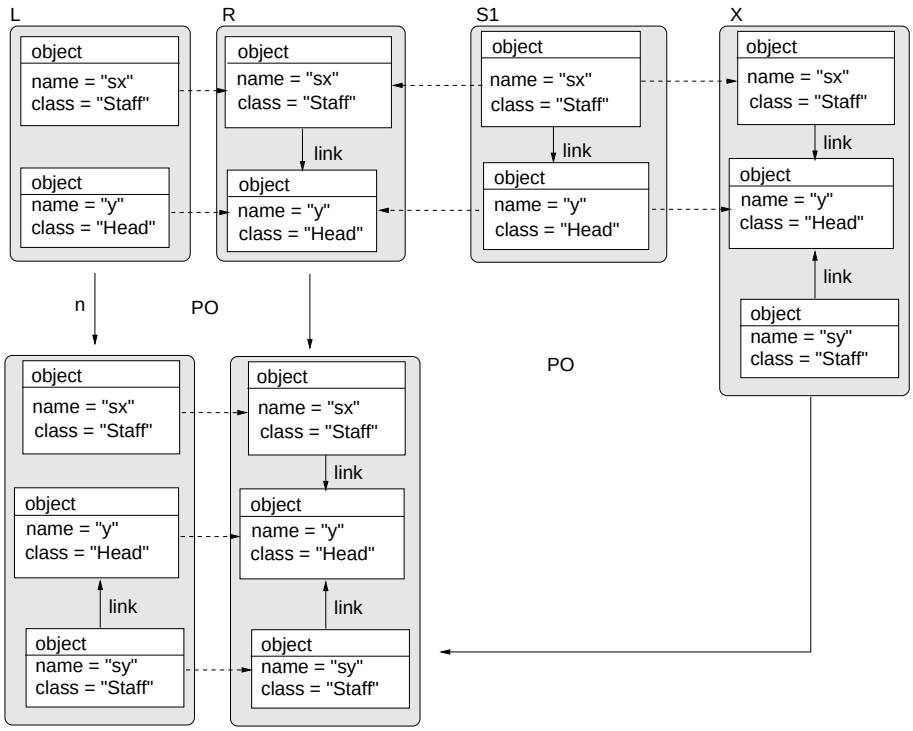

Fig. 18. *Example for an extending rule and a negative constraint.*

and `Head` roles and the negative constraint which forbids two staff members in the `Head` role. The overlap S is one of two possible overlaps, since the link must be included in S by construction 3. The application condition n forbids the assignment of a staffer to the role `Head` if there is already a staffer in this role.

The second example in figure 19 shows the deleting rule *destroy permission* for a `CISPermission` and the positive constraint which requires a `CISPermission` in each `CISView` when assigned to role `Nurse`. The overlap S is the only possible one in this example. The application condition n forbids to delete the permission object when there exists a connected `CISView` assigned to a role `Nurse`.

The extended graph rules can be translated back into a UML representation. We need only an additional UML representation for application conditions. We choose

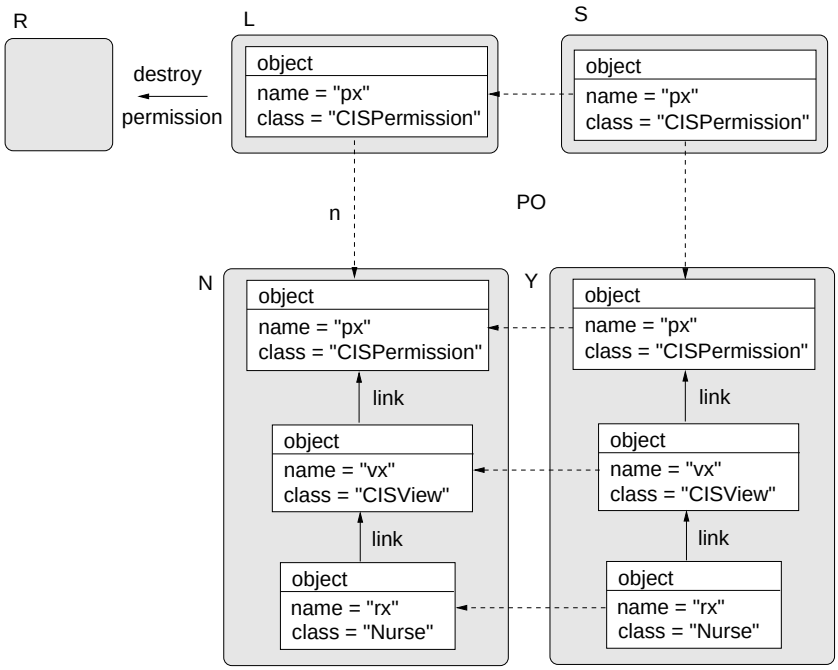

Fig. 19. *Example for deleting rule and positive constraint.*

the stereotype `<<not>>` (fig. 20 shows the UML representation of the example in figure 18). The intended meaning of an object diagram for a policy rule with stereotype `<<not>>` is, that the rule can be applied when all objects/links without stereotype `<<create>>` occur and all objects/links with stereotype `<<not>>` do not occur.

9 Related Work

Recent research concerns the integration of security engineering into the software development process. Since the UML is the de-facto standard modeling language in practice, the approaches try to integrate security aspects into the UML.

Epstein and Sandhu introduce in [10] a UML-based notion for RBAC. The notion, however, is neither suitable for the verification of security properties nor for the generation of access control specifications. Quoting from the paper: "Although there is a check on the UML syntax, there is no logic or semantic check. We have to trust he designer to accurately depict the model" and "...this paper ...does not show a methodical approach for defining constraints for UML or for the RBAC model...."

Jürjens extends in [14] the UML for specifying aspects of multi-level secure systems and security protocols. He proposes a tailored formal semantics to formally

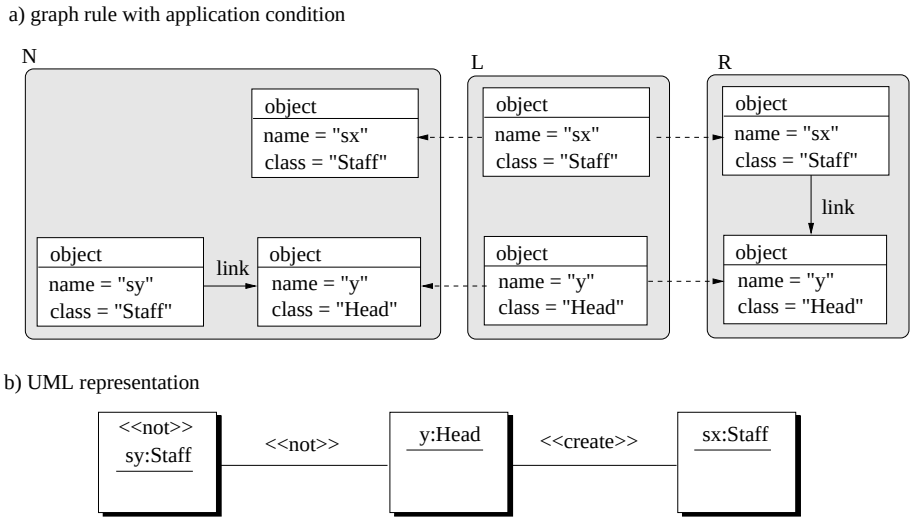

Fig. 20. *UML representation of application conditions.*

evaluate UML diagrams and to indicate possible weaknesses. In contrast to our approach, there is no model-driven *integration* of security and the verification does not provides an automatic consistency construction to resolve unsatisfied security constraints.

Fernandez-Medina et al. extend in [11] the UML to support the design of secure databases. They consider a multilevel database model in which all model elements have a security level and a security role which are both modeled by tagged values. They extend the OCL to be able to represent multilevel system constraints. Unlike [11] which is developed for multilevel databases, our approach is more general for any access control model and not restricted to databases. In contrast to [11] which extends the OCL by non-standard features, we use for the specification of constraints the standard OCL. The OCL extension has the advantage of a clearer and more legible specification in the specific context of multilevel databases, but does not ensure compatibility.

An approach closely related to ours is SecureUML [19]. SecureUML is a UML-based modeling language for the model-driven development of secure systems. It provides support for specifying constraints, as well. The security information integrated in the UML models is used to generate access control infrastructures. In contrast to our approach, SecureUML focuses on static design models which are closely related to implementations. Therefore, there is no support for detecting security requirements (e.g., which roles are needed and which permissions they need). Unlike our approach, which is suitable for arbitrary access control models, SecureUML builds

on RBAC. Moreover, SecureUML does not have a formal semantics to verify security properties.

10 Conclusion

We have presented an approach to the specification of AC control policies in UML by means of UML class and object diagrams that can be modelled with existing UML tools. A translation of the UML AC specification into a graph-based security framework permits the application of verification concepts from graph transformations to reason about the coherence of a UML AC specification.

The UML AC specification can be modelled using existing UML CASE tools. One aim of future work is the transformation of the XMI export (got by the CASE tool) into a XML format for graphs [24]. Then, graph transformation tools [9] can be used for the automatic verification of AC requirements.

References

1. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of ocl using collaborations. In *Proc. of UML 2001 - The Unified Modeling Language*, number 2185 in LNCS, pages 257–271. Springer, 2001.
2. G. Brose. A typed access control model for CORBA. In *Proc. ESORICS*, LNCS 1895, pages 88–105. Springer, 2000.
3. G. Brose. *Access Control Management in Distributed Object Systems.* PhD thesis, Freie Universität Berlin, 2001.
4. G. Brose. Raccoon — An infrastructure for managing access control in CORBA. In *Proc. Int. Conference on Distributed Applications and Interoperable Systems (DAIS).* Kluwer, 2001.
5. G. Brose, M. Koch, and K.-P.Löhr. Integrating Access Control Design into the Software Development Process. In *Proc. of 6th International Conference on Integrated Design and Process Technology (IDPT)*, 2002.
6. G. Brose, M. Koch, and K.-P. Löhr. Entwicklung und Verwaltung von Zugriffsschutz in verteilten Objektsystemen – eine Krankenhausfallstudie. *PIK – Praxis der Informationsverarbeitung und Kommunikation*, (1/03), 2003.
7. P. T. Devanbu and S. Stubblebine. Software engineering for security: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering.* ACM Press, 2000.
8. D. D'Souza and A. Wills. *Components and Frameworks: The Catalysis Approach.* Addison-Wesley, 1998.
9. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. II: Applications, Languages, and Tools.* World Scientific, 1999.
10. P. Epstein and R. Sandhu. Towards A UML Based Approach to Role Engineering. In *Proc. of ACM RBAC*, 1999.
11. E. Fernandez-Medina, A. Martinez, C. Medina, and M. Piattini. Uml for the design of secure databases: Integrating security levels, user roles, and constraints in the database design process. In Jürjens et al. [13], pages 93–106.

12. R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars - a constructive approach. In *Proc. SEGRAGRA'95 Graph Rewriting and Computation*, number 2. Electronic Notes of TCS, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.
13. Jürjens, Cengarle, Fernandez, Rumpe, and Sandner, editors. *Critical Systems Development with UML*, number TUM-I0208 in Technical Report TU München, 2002.
14. J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proc. of UML 2002*, number 2460 in LNCS, pages 412–425. Springer.
15. J. Jürjens. Towards Development of Secure Systems Using UMLsec. In *Proc. of FASE'01*, number 2029 in LNCS, pages 187–200. Springer, 2001.
16. M. Koch, L. Mancini, and F. Parisi-Presicce. Foundations for a graph-based approach to the Specification of Access Control Policies. In F.Honsell and M.Miculan, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, Lect. Notes in Comp. Sci. Springer, March 2001.
17. M. Koch, L. Mancini, and F. Parisi-Presicce. Conflict Detection and Resolution in Access Control Specifications. In M.Nielsen and U.Engberg, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2002)*, Lect. Notes in Comp. Sci., pages 223–237. Springer, 2002.
18. M. Koch, L. Mancini, and F. Parisi-Presicce. Decidability of Safety in Graph-based Models for Access Control. In *Proc. of 7th European Symposium on Research in Computer Security (ESORICS)*, 2002. to appear.
19. T. Lodderstedt, D. Basin, and J. Doser. SecureUML:A UML-Based Modeling Language for Model-Driven Security. In *Proc. of 5th Int. Conf. on the Unified Modeling Language*, number 2460 in LNCS. Springer, 2002.
20. OMG. *CORBA 3.0 New Components Chapters, TC Document ptc/99-10-04*. OMG, Oct. 1999.
21. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
22. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role–based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.
23. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0, Final Draft*, Oct. 2000.
24. G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In *Proc. of Uniform Approaches to Graphical Process Specification Techniques UNIGRA'01*, number 47 in ENCS, 2001.
25. A. Tsiolakis. Consistency Analysis of UML Class and Sequence Diagrams based on Attributed Typed Graphs and their Transformation. Technical Report 2000/3, TU Berlin, March 2000.