Actor-Centric Modeling of User Rights

Ruth Breu[1] and Gerhard Popp[2]

[1] Research Group "Quality Engineering", Universität Innsbruck
Institut für Informatik, A-6020 Innsbruck, Austria
Ruth.Breu@uibk.ac.at

[2] Software & Systems Engineering, Technische Universität München
Institut für Informatik, D-85748 Garching b. München, Germany
Gerhard.Popp@in.tum.de

Abstract. In this paper we present a novel approach for the predicative specification of user rights in the context of an object oriented use case driven development process. We extend the specification of methods by a permission section describing the right of some actor to call the method of an object. Moreover, we introduce a representation function that describes how actors are represented internally in the system. As syntactic and semantic framework we use a first-order logic with a built-in notion of objects and classes provided with an algebraic semantics. We demonstrate that our approach can be realised in OCL.

1 Introduction

The requirement of protecting data from unauthorised user access is as old as multi-user computing. Applications such as ERP systems or health information systems with hundreds or thousands of users handling sensible data offer sophisticated mechanisms for rights modeling. With the new web technologies the importance of data protection mechanisms will even grow. The more companies will open their core business processes to external partners the more important the enforcement of access rights will become.

Data protection is intimately connected with two aspects – authentication on the one hand side and access control on the other side. Authentication aims at identifying actors (persons or external systems) interacting with the system. Access control is concerned with the protection of information resources.

With the prominent RBAC model an adequate paradigm for implementing access rights has been developed [1,2,3]. Access rights in this model do not adhere to single users but are associated with *roles*. Basically each user may have one or several roles and each role is associated with a set of *permissions*, where each permission defines the kind of access (the operation, e.g. read, update) to some object. Today the RBAC model is one of the most established access models. This model is supported by many systems like operation systems, data bases and middleware platforms.

Despite of the role paradigm, data access control remains a complex task in real applications. In particular, data access control in most cases concerns different layers ranging from the user interface and the application layer to the database. Moreover, upcoming inter-organisational applications require novel (credential-based) techniques for enforcing user rights [4].

Despite the complexity of the task, an aspect neglected so far in the literature has been the analysis and design phase of user access models. Developing user right concepts for applications in areas like health care, e-government or knowledge management requires both an implementation-independent analysis framework and a step-by-step approach. Moreover, since user access models have to be developed in close cooperation of system designers and clients user rights modeling has to be integrated into the requirements engineering process.

In this paper we present an approach to the specification of user access rights satisfying these needs. In particular, our approach is based on the following three basic ideas.

First, we conceive user rights modeling as a task within the context of an object oriented development method such as the Unified Process [5] or the V-Model [6]. This means that our method is entirely integrated within the context of business process modeling, use case modeling and systems analysis. More generally, the approach we present in this paper is part of a process model for security engineering [7]. This process model extends an object oriented kernel model by techniques, artifacts and activities supporting the systematic construction of security-critical systems.

Second, our method supports the stepwise development of user right models. This ranges from informal textual statements to a complete predicative specification. The implementation independent specification of user rights has several advantages. The most important one is that the model developed is a compact and concise representation of knowledge involving many parts of the implementation. Moreover, complete user right models have the potential to be automatically transformed into code.

Third, our approach is provided with a formal semantics in an algebraic setting. On the syntactic side we extend the specification of each method by a *permission section* describing for each actor (or role) if this actor has the right to call this method on an object of the given class. Since actors play a central role for the specification of access permissions we call our method *actor-centric*. The permission is described by a first-order predicate over a structure with a built-in notion of objects or, in the context of UML, by an OCL statement [8,9].

Additionally, we introduce a representation function to describe how actors are represented internally in the system. This representation function is an abstraction of the authentication procedure in the implementation and allows specifications of the kind *"the user has the right to view his/her own data"*. The semantics of permissions and of the representation function can be embedded in a straight forward way in the algebraic theory presented in [10,11].

Related work in the literature mainly deals with the RBAC approach (e.g. [1,2,3]). Our approach goes beyond in several respects. First, we are concerned with the development process of user right models in the context of object oriented modeling techniques. Moreover, we provide an increased expressiveness by supporting arbitrary first-order predicates to specify user rights.

An approach with similar expressiveness is [12]. While this approach has been designed with the primary goal of code generation, our focus has been the development of concepts adequate for the whole development process. A scheme of user rights modeling in the context of use cases has been first presented by [13]. We overtake some of these ideas, but present a more elaborate theory. Further references in a specific setting are [14,15] dealing with the development process and checking of user rights in SAP applications.

The rest of this paper is organised as follows. In section 2 we give some background information. Section 2.1 gives an overview of the artifacts of a kernel object oriented method our approach is based on. Section 2.2 sketches the syntactical and semantical specification framework. A formal model of user rights is given in section 3, which is divided into a description of the representation function (see section 3.1) and permissions (section 3.2). The specification of permissions in OCL is presented in section 4 and extensions are introduced in section 5. In section 6 a conclusion is drawn.

In the sequel we assume the reader to be familiar with basic concepts and notations of object oriented modeling with UML and OCL.

2 Basics

In this section we first give a short overview of the modeling context our approach is based on. Throughout this paper we will use as running example a case study based on *TimeTool*, a software project that was realised in our research group. *TimeTool* is a portion of a project management tool allowing team workers to account worked hours for projects and allowing project managers to supervise project budgets.

2.1 The Core Object Oriented Artifacts

In this section we will shortly characterise the core artifacts of the object oriented process together with the dependencies.

The Business Model captures the organizational environment of the IT-system. The model describes

- Who (or, more precisely, what roles) act in the business domain (*actors*)
- What *activities* the actors perform
- Which *objects* the activities need as input and which objects they produce as output

In *TimeTool* the actors are the *project manager*, the *team worker* and the *administrator*. Example activities are *Account Worked Hours* and *Post Adjustment* (performed by the team worker or the project manager) and *Prepare Monthly Report* (performed by the project manager). Example classes in the application domain are the *project*, the *accounting* and the *team worker.*

Actors, activities and objects are modeled by several diagrams. In the UML context this comprises *activity diagrams* modeling business processes and activities, and *class diagrams* modeling the organizational structure of the company and the structure of static concepts (users, projects, accountings, etc.).

The System Requirements describe a black box view of the system to be developed based on the concept of *use cases*. Each use case corresponds to a coherent interaction between some actor and the system. The use cases together describe the whole functionality of the system. Basic concepts of the use case model are

– the *actors* interacting with the system
– the *use cases*
– the *objects* being involved in the use cases

In extension to business modeling the actors in the use case model might not only represent human beings (like the *team worker* and the *project manager*) but also external systems. For instance, additional actors in *TimeTool* are the web browser and the human resources system which hosts the databases with available staff and to which the system connects to for data import. Sample use cases are *Account Worked Hours*, *Adjustment Posting* and *View Statistics*.

The System Requirements basically consist of two descriptions, the *use case diagram* together with the *textual description* of use cases, and the *class model* describing the static concepts.

Commonly, the class model of the System Requirements is the same or a refined version of the class model of the Business Model. Figure 1 depicts the class model for *TimeTool*.

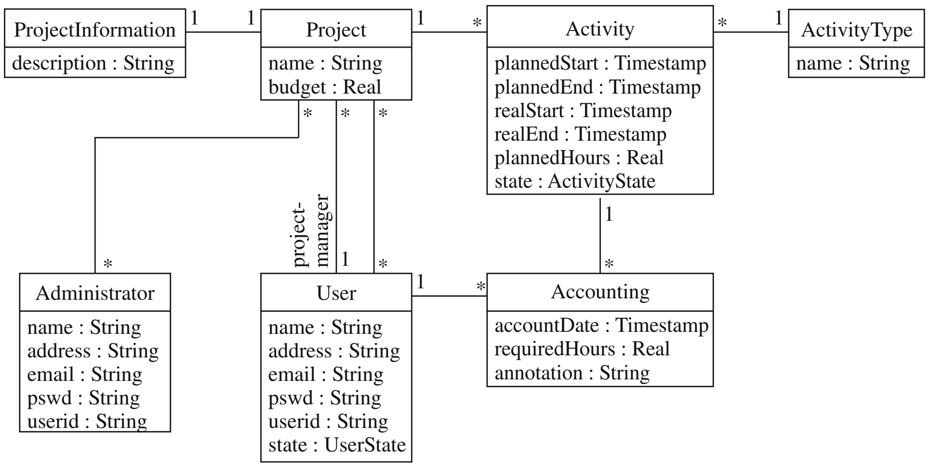


Fig. 1. Class Diagram of the *TimeTool* Example

The Application Architecture refines the level of description. More precisely, the system is divided into a set of *logical components*. Each component is responsible for a portion of the system structure and behaviour. Interfaces enable the independent development of the system components.

Concerning the design of the system behavior the textual descriptions of the use cases are refined into *scenarios* in the Application Architecture. The scenarios describe the use cases as message flows between objects. That way, an object oriented view of the whole system is achieved. In the UML context the Application Architecture typically consists of the following diagram types: One or several *component diagrams* containing components, interfaces and the related classes, a set of *sequence diagrams* each one related to some use case, and *state diagrams* modeling complex processes or class interfaces.

Business Model, System Requirements and the Application Architecture have in common that they rely on an application oriented system view and are independent of any technical platform.

The Software Architecture is based on an implementation oriented view of the system. In this model the hardware and software platform is chosen and roughly described. Since we will not deal with the platform dependent level in this paper we do not go into more detail at this place.

2.2 The Specification Framework

As specification framework we use the specification language P-MOS [10,11]. P-MOS supports first-order predicates with a built-in notion of objects and is provided with a semantics in an algebraic setting. The kernel syntactic constructs can be found below.

P-MOS can be compared in its expressiveness with OCL but provides the full flexibility of an algebraic specification language. In our approach P-MOS serves as an intermediate language for developing concepts and providing a semantics. As we will demonstrate OCL can be used as target language within our method.

P-MOS Expressions. Each P-MOS expression is based on a class diagram. The expression describes a navigation in an object structure delivering some result.

Semantically each P-MOS expression is interpreted in the context of a so-called object environment describing a concrete object structure over the class diagram given. P-MOS is a hybrid language which means that we distinguish between basic types and class types. While an expression of some basic type (such as *Bool* or *int*) denotes a value (*true*, *false*, *0*, *1*, ...) an expression of a class type denotes a reference to an object in the given object structure.

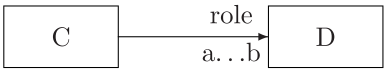
P-MOS expressions are built by the application of one of the six rules that can be found below. We assume to be given a set of basic data types (such as *Bool* and *int*) and type constructors. In particular, we assume a type constructor *Set[_]* describing sets of arbitrary elements.

(1) Basic Functions. Let f: $(s_1, \ldots, s_n)$ s be a function in a basic data type (such as *+: (int, int) int* or *true: () Bool*). If $e_1, \ldots, e_n$ are P-MOS expressions of type $s_1, \ldots, s_n$ then $f(e_1, \ldots, e_n)$ is a P-MOS expression of type s.

(2) Variables. Let X be a set of typed variables. Then each variable x of type s is a P-MOS expression of type s.

(3) Attributes. Let *A:T* be some attribute of class C (*T* either being some class name or some basic data type) and *e* be some P-MOS expression of type *C*. Then *e.A* is a P-MOS expression of Type *T*.

(4) Assocations. Let us assume an assocation



in the class diagram. Then *e.role* is a P-MOS expression of type *Set[D]* (*D*, resp. in the special case a...b = 1...1) if *e* is expression of type *C*. If the role name is missing then the navigation expression is constructed by *e.d* (the class name *D* written in lower case).

(5) Generalisation. If *e* is P-MOS expression of type *C* and *C* is subclass of class *D* (in the transitive closure) then *e* is P-MOS expression of type *D*.

(6) State-Based Functions. Let ***funct*** *f:*$(T_1,\ldots,T_n)$ *T* be a state based function (where $T_1,\ldots,T_n$, *T* are either basic types or classes) and $e_1, \ldots, e_n$ are P-MOS expressions of type $T_1,\ldots,T_n$. Then $f(e_1, \ldots, e_n)$ is a P-MOS expression of type *T*.

A state-based function describes a generic navigation in the given object structure based on *n* parameters and delivering a result of type *T* (either a basic value or some object reference). The properties of a state-based function are defined by P-MOS predicates as defined below.

Query operations in the class diagram include state-based functions in the following way:

For each query operation $f{:}(x_1{:}T_1,\ldots,x_n{:}T_n){:}T$ in class C we define a state based function *funct* $f{:}(C,T_1,\ldots,T_n)$ *T* and write expressions $f(e,e_1, \ldots, e_n)$ as $e.f(e_1, \ldots, e_n)$ in the usual way.

P-MOS Predicates. Based on the notion of P-MOS expressions P-MOS predicates are formed in the usual way as given in table 1.

Table 1. P-MOS Predicates

P-MOS Predicate	Element Description
e1 = *e2*	*e1*, *e2* P-MOS expressions of the same type
$\neg P$, $P1 \vee P2$, $P1 \wedge P2$, $P1 \Rightarrow P2$, $P1 \Leftrightarrow P2$	*P*, *P1*, *P2* predicates
$\forall x{:}T.P$, $\exists x{:}T.P$	*P* predicate, *x* variable, *T* type expression

Table 2. Access Rights from Team Workers and Project Managers

actor → ↓ *class*		**Team Worker**		**Project Manager**
Accounting	R: W/C:	all own accountings (independent of the project) own accountings from released activities	R/W/C:	all accountings of activities of own projects (i.e. where actor is the project manager of)
Activity	R: W/C:	all activities from projects allocated to the actor –	R/W: C:	all activities of own projects –
Project	R: W/C:	all projects –	R: W/C:	all projects –
User	R: W/C:	all users –	R: W/C:	all users –

3 Formal Modeling of User Rights

The central notion for capturing individuals and their roles in business process modeling and use case modeling is that of an *actor*. For instance, in the business process model an actor stands for the person (or, more precisely, for the role of this person) being involved in the business process. In extension, actors in the use case modeling also may stand for the roles external systems play.

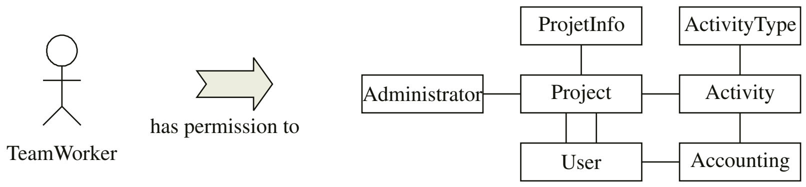


Fig. 2. Actors have Permissions on Objects.

The key idea to the modeling of user rights in our approach is that actors have some kind of permissions with respect to objects of the class model (see Figure 2). In this respect our user right model both refers to the model containing the actor (business process model or use case model) and to the class model. The separation of role concept and classes has the advantage that the way how roles are represented in the system has not to be fixed within requirements elicitation.

In early phases of the development we may wish to specify user rights in an informal, textual way. Table 2 depicts such a textual user rights model for our case study. The informal model contains coarse-grained permission categories

(R = Read, W = Write, C = Create) for each class. The textual statements characterise the objects of the given class which the actor may read, write or create. For large parts of an application such a coarse-grain description may be sufficient. For critical parts we require more fine-grained ways to express user rights. For that reason we offer a specification mechanism at the level of methods. More precisely, each *method* m in *class* C is associated with a *permission*

$$perm_{C,m}$$

specifying under which condition an actor has access to call the method on an object of the given class. In section 5 we will present a mechanism to aggregate permissions supporting a more coarse-grained level of detail.

What is missing in the framework sketched so far is a connection between actors and classes. Such a connection is required in cases where permissions refer to the actor himself like in the example the *team worker has read permissions to own accountings.* In order to support such kinds of specifications we provide a function

$$rolerep$$

mapping actors to objects of some class. This class (in most cases some class like *User*) is the internal representation of actors. In fact the representation function is an abstraction of the authentication procedure in the implementation. More information on that will be given in section 3.1.

To conclude this section we shortly summarise in table 3 the development steps of a user rights model in the context of the object oriented process.

Table 3. Development Steps of a User Rights Model

Development Step	**Activities**
Business Process Model	Informal description of actor permissions in tables.
Use Case Model	Adaptation of the informal model to the actors of the use case model (e.g. including extended systems). If possible development of a first formal model.
Application Architecture	Development of a complete formal model.

In the context of iterative development the abstract user right model has to be adapted and extended in each iteration (e.g. concerning new classes).

In the sequel we will present the formal mechanism of method permissions (section 3.1) followed by the representation function (section 3.2).

3.1 The Function *rolerep*

As motivated in the preceding section the function *rolerep* maps actors to their internal representation. In order to provide a homogeneous specification framework within P-MOS we internally extend the class diagram by a class hierarchy representing the actor roles.

In particular, we introduce a superclass *ACActor* modeling all kinds of actors. Subclasses of the class *ACActor* are all actors defined in the business process or use case model, respectively. Actor hierarchies in the use case model are transformed in a corresponding class hierarchy. In the example, we obtain subclasses *ACAdministrator*, *ACTeamWorker* and *ACProjectManager*, where *ACProject-Manager* is a subclass of *ACAdministrator* (see Figure 3). In our model actors

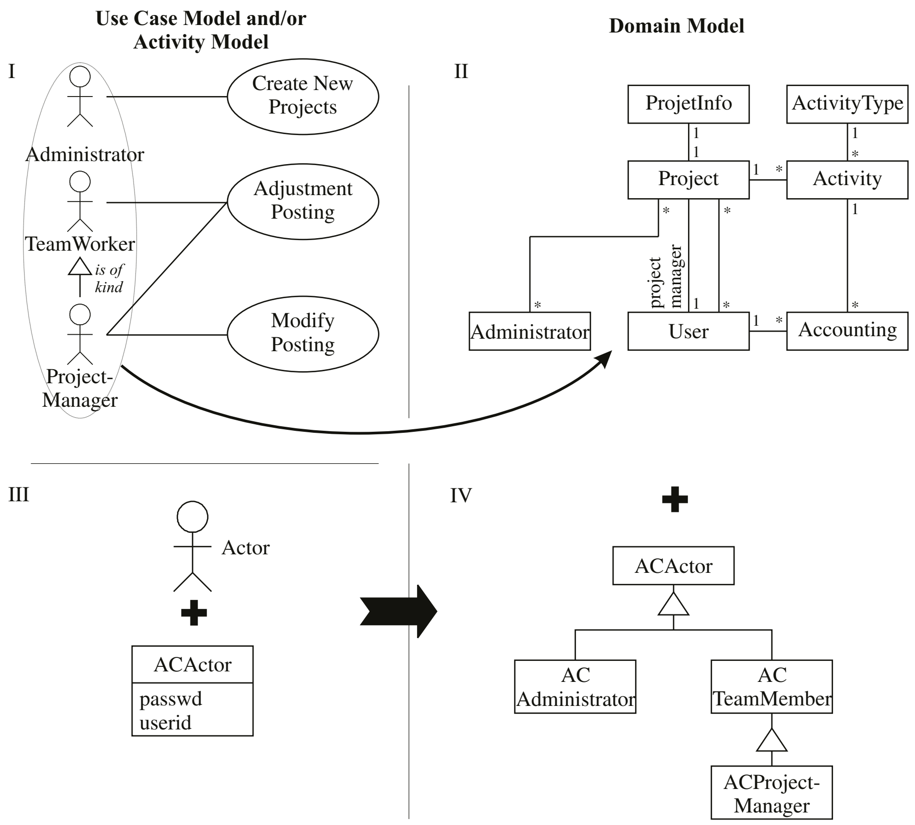


Fig. 3. The Extension of Actor Classes for an Actor–User Mapping.

also may have attributes. These attributes represent the input that is required to authenticate the actor (human being or external system) within the system. In most cases the attributes are the userid and the password, but also biometric data, credentials or “no information” (if the actor is anonymous to the system) are possible.

Formally, the (state-based) representation function *rolerep* has the functionality

$$\textbf{\textit{funct}}\; rolerep : (ACActor)\, Object$$

where *Object* is the superclass of all classes (like in Java).

The function *rolerep* is part of the actor class $ACActor$ and can be specified in this class for all actor types, or in each subclass for specific actor types. Equivalently the function *rolerep* can be conceived as query operation of functionality $rolerep : ()\, Object$.

As example we present the specification of the *rolerep* function for the two actors *TeamWorker* and *Administrator* of the *TimeTool* example.

ACTeamWorker

passwd: String
userid: String

rolerep : () Object

$$\forall\, tw : ACTeamWorker\ .\ \forall\, u : User\ .\ tw.rolerep\,() = u \ \Rightarrow$$
$$u.state = \sharp active \ \wedge$$
$$tw.userid = u.userid \ \wedge\ tw.passwd = u.passwd$$

ACProjectManager

rolerep : () Object

$$\forall\, pm : ACProjectManager\ .\ \forall\, u : User\ .\ pm.rolerep\,() = u \ \Rightarrow$$
$$\exists\, p : Project\ .\ p.projectmanager = u$$

The specification of actors including the representation function in our modeling framework is part of the actor description in the use case model. During construction the representation function is implemented by the authentification procedure.

3.2 Permissions

Permissions are method preconditions associated with the semantics that the corresponding method can only be executed if the permission expression is evaluated to true at the beginning of the execution. Since the basis for our approach is the *fail-safe defaults principle*, every method execution which is not permitted explicitly through a permission is prohibited. Each method permission may depend on the calling actor, on the actual object, and on the actual parameters of the method call. Thus, permissions are state-based functions of the kind

$$\boldsymbol{funct}\; perm_{\,C,m} : (ACActor, C, T_1, \ldots, T_n)\; Bool$$

where C is a class, and m a method in C of the form *m-id*: $(x_1 : T_1, \ldots, x_n : T_n)\, T$ (In the special case of create methods or class methods the parameter of type C is omitted).

The properties of method permissions are specified by P-MOS predicates describing conditions over the given object structure.

In the following we will give a few examples of permission specifications. We model some access rights from Table 2 for the *team worker* and the *project manager* of our *TimeTool* example.

Example 1a: As first example we specify the permission that a *team worker* can read all his own accountings, independent of the project the accounting belongs to. As sample we use the method *getAccountingDate()* as representative for a reading method.

$$\begin{aligned}&\forall\, tw : ACTeamWorker\;.\;\forall\, a : Accounting\;.\\&a.user = tw.rolerep\,()\\&\quad\Rightarrow\;perm_{\,Accounting,\,getAccountingDate}(tw, a)\end{aligned}$$

The expression states, that the user object, that is linked with the accounting object must be the internal representation of the given *team worker*. Only if this expression evaluates to *true*, the *team worker* has access to the *getAccounting-Date()* method of the class *Accounting*.

Example 1b: As second example we consider the permission that a *project manager* can read all accountings associated with his own projects. Again we specify the permission of *getAccountingDate*.

$$\begin{aligned}&\forall\, pm : ACProjectManager\;.\;\forall\, a : Accounting\;.\\&a.activity.project.projectmanager = pm.rolerep\,()\\&\quad\Rightarrow\;perm_{\,Accounting,\,getAccountingDate}(pm, a)\end{aligned}$$

Example 2: A further permission for the *team worker* is, that he can only write accountings to released activities. Activities are associated with a state which may be set to *released* or *frozen*. In this way, it can be prohibited that somebody manipulates accounting objects related to finished activities. The permission is given in the following predicate:

$$\begin{aligned}&\forall\, tw : ACTeamWorker\;.\;\forall\, a : Accounting\;.\\&a.user = tw.rolerep\,()\;\wedge\\&a.activity.state = \sharp released\\&\quad\Rightarrow\;perm_{\,Accounting,\,writeAccountingDate}(u, a)\end{aligned}$$

Example 3: As last example we study the permission of a create method. The *project manager* can only create accountings from activities of own projects. Here we assume for short, that the create method has the activity and the user the accounting refers to as only attributes.

$$\begin{aligned}&\forall\, pm : ACProjectManager\;.\;\forall\, ac : Activity\;.\;\forall\, u : User\;.\\&ac.project.projectmanager = pm.rolerep\,()\\&\quad\Rightarrow\;perm_{\,Accounting,\,create}(pm, ac, ac)\end{aligned}$$

Permission Inheritance. There are two aspects where the specification of permissions interferes with the concept of ineritance.

The first aspect is related with the hierarchy of actors (see quarter IV of Figure 3). In our example the *project manager* is a special kind of *team worker*. Thus, for permissions of a *project manager* not only the axioms for *project managers* but also for *team workers* hold. For instance, referring to Example 1 and 2, a *project manager* has reading access both to his own accountings (independent of the projects) and to all accountings of his own project (independent of the user).

The second aspect is related with inheritance in the domain model referring to the objects that we want to protect with our permissions. In the same way as above, methods of subclasses inherit the permissions of their superclasses. In addition, we provide the possibility to let permissions unspecified in superclasses and deferring their specification to their subclasses.

4 Specification of Access Policies in OCL

Regarding tool support for the modelling of access rights and for implementation aspects the formal specification in the framework presented has to be transformed into a specification language, designed especially for use in the context of diagrammatic specification languages such as UML.

In the following we show the realisation of our concepts within the Object Constraint Language (OCL) [8,9] that is part of UML. We extend the specification section of a method (comprising the pre- and postcondition) by an additional *permission section*. In this section the method permission is specified by an OCL-expression using the variables of the method, the actual object and the actor which is handled as parameter of the permission section.

The representation function *rolerep* is treated as query operation of the actor hierarchy as introduced in section 3.1 and may be used in the permission section.

Example 1: A team worker can read all his own accountings, independent of the project the accounting belongs to (exemplified by the permission of *getAccountingDate*).

```
context Accounting :: getAccountingDate()
   perm (act : ACTeamWorker):
      self.user = act.rolerep()
```

Example 2: A team worker can only write own accountings of released activities.

```
context Accounting :: writeAccountingDate()
   perm (act : ACTeamWorker):
     self.user = act.rolerep() and
     self.activity.state = ActivityState::released
```

Example 3: The project manager can only create accountings for activities of own projects, i.e. activities of projects, where he is project manager of.

```
context Accounting :: create(a : Activity, u : User)
   perm (act : ACProjectManager):
     a.project.projectmanager = act.rolerep()
```

Regarding the semantics of a method permission it has to be made clear that the given actor is *not* the object directly calling the method but the role initiating the call of this method from outside the system (eventually causing a chain of method calls). In the implementation the calling actors can be handled by an additional method parameter or, like in J2EE, by some method call infrastructure.

5 Extensions

As explained in the previous section our basic view of user rights is that of an actor having permission to perform a certain method on a certain object. This fine-grained paradigm provides a maximum of flexibility for specifying any kind of user permissions in all phases of the development. However, it is clear that for practical applications we need an aggregation mechanism for supporting more coarse-grained specifications. We therefore introduce the notion of *method categories*. A method category CAT basically is a set of methods

$$CAT \subseteq METH$$

where $METH$ is the set of all methods (sorted by the classes they belong to) in the system. Categories may contain methods of a single class (we use the class name as index in this case) or of several classes. Moreover, categories may comprise other categories, i.e. categories may be structured in a hierarchical way. We define coarse-grained permissions

$$perm_{Cat_C} : (ACActor, C)\, Bool$$

for each category Cat_C containing methods of class C. A category permission induces method permissions in the obvious way.

$$\forall\, a : ACActor,\, o : C,\, a_1 : T_1, \ldots, a_n : T_n\ .$$
$$perm_{Cat_C}(a, o) \Rightarrow perm_{C,m}(a, o, a_1, \ldots, a_n)$$

for all methods m of class C in Cat_C. Of course, such a category permission may only depend on the actor and the actual object.

If a category Cat comprises methods of different classes (and create methods) we define permissions.

$$perm_{Cat} : (ACActor)\, Bool$$

and induce the following method permissions for all methods m of class C in Cat.

$$\begin{aligned}&\forall\, a : ACActor,\ o : C,\ a_1 : T_1, \ldots, a_n : T_n\ .\\ &perm_{\,Cat}(a) \Rightarrow perm_{\,C,m}(a, o, o_1, \ldots, o_n)\end{aligned}$$

Category permissions of this kind only depend on the actor initiating the method call. Concerning the application of this concept we provide a set of predefined categories:

$READ_C$	the category of all methods reading some attribute of class C
$UPDATE_C$	the category of all methods updating some attributes of class C
$CREATE_C$	the category of all creation methods of class C

Moreover, we define the get and set methods of attributes to be predefined members of the $READ_C$ and $UPDATE_C$ category, respectively. Please notice that there may be methods both belonging to the $READ_C$ and the $UPDATE_C$ category.

As an example the informal specifications of Table 2 can be immediately expressed in a corresponding way with method categories. E.g. the clause *"The team worker can read all own accountings"* can be expressed with the following category permission:

$$\begin{aligned}&\forall\, tw : ACTeamWorker\ .\ \forall\, a : Accounting\ .\\ &a.user = tw.rolerep\,()\\ &\quad \Rightarrow perm_{\,READ_{Accounting}}(ac, a)\end{aligned}$$

The set of predefined method categories may be replaced and complemented by user-defined categories. This is advisable if a whole part of the class diagram is associated with the same kind of permissions (e.g. the permission *true*). A further typical case in which we need a more fine-grained categorisation is the following. The attributes of a class (e.g. *Person*) are divided into critical (e.g. salary of a person) and uncritical ones (e.g. name and address of a person).

6 Conclusion

In the preceding sections we introduced a formal specification framework for the modeling of user rights. Our method is novel in the respect that it is completely integrated in the concepts of use case driven object oriented modeling and provides the full expressiveness of first-order logic. We separate the aspects of authentication and data access and thus enable a concise specification of permissions connecting roles and their internal representation. We both support the specification of permissions on the most fine-grained level of methods and on a coarse-grained level based on the notion of method categories.

Currently we conduct two case studies in real contexts (health information systems, e-government) in order to validate our approach. Future work will be done in several directions. First of all we will develop tool support for our method. This comprises the possibility to develop method permissions, actor specifications and the definition of categories and category permissions within some UML tool. Moreover, our concept of specifying permissions well be applied in implementation oriented contexts. In the project SECTINO we develop a framework for specifying access policies for inter-organisational workflows based on Web Services. Besides this, we work on a testing environment testing and analysing rights in collaborative systems.

References

1. Ferraiolo, D.F., Chandramouli, R., Kuhn, D.R.: Role-Based Access Control. first edn. Artech House Publishers (2003)
2. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-Based Access Control. In: ACM Transactions on Information and System Security. Number 3. ACM (2001) 224–274 http://csrc.nist.gov/rbac/rbacSTD-ACM.pdf.
3. Sandhu, R.S.: Role Hierarchies and Constraints for Lattice-Based Access Controls. In: Proceedings of the European Symposium on Research in Security and Privacy. (1996)
4. Miller, J., Fan, M., Sheth, A.P., Kochut, K.: Security in Web-Based Workflow Management Systems. In: Proceedings of the International Workshop on Research Directions in Process Technology, Nancy, France (1997)
5. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison Wesley Longman, Inc. (1999)
6. http://www.v-modell.iabg.de.
7. Breu, R., Burger, K., Hafner, M., Jürjens, J., Popp, G., Wimmel, G., Lotz, V.: Key Issues of a Formally Based Process Model for Security Engineering. In: Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA03), Paris, December 2 - 4, 2003. (2003)
8. Warmer, J., Kleppe, A.G.: The Object Constraint Language – Precise Modeling with UML. first edn. Addison Wesley Longman, Inc. (1999)
9. OMG: Unified Modeling Language Specification – Version 1.5 (2003)
10. Breu, R.: An Integrated Approach to Use Case Based Development (2004) To appear.
11. Breu, R.: Objektorientierter Softwareentwurf – Integration mit UML. Springer-Verlag (2001) in German.
12. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-based Modeling Language for Model-Driven Security. In: Proceedings LNCS 2460, Springer (2002) 426–441
13. Fernandez, E., Hawkins, J.: Determining role rights from use cases. In: Workshop on Role-Based Access Control, ACM (1997) 121–125
14. Höhn, S., Jürjens, J.: Automated Checking of SAP Security Permissions. In: Proceedings of the 6th IFIP WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS), Nov. 13-15, 2003, Lausanne, Switzerland, Kluwer (2003)
15. Services, I.B.C.: SAP Berechtigungswesen, Design und Realisierung von Berechtigungskonzepten f r SAP R/3 und SAP Enterprise Portal. SAP Press (2003) in German.