Articulating and Enforcing Authorisation Policies with UML and OCL

Karsten Sohr
Department of Mathematics and Computer Science
Universität Bremen
Bibliothekstr. 1
28359 Bremen, Germany
sohr@tzi.de

Gail-Joon Ahn*
Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC 28223, USA
gahn@uncc.edu

Lars Migge
Department of Mathematics and Computer Science
Universität Bremen
Bibliothekstr. 1
28359 Bremen, Germany
lmigge@tzi.de

ABSTRACT

Nowadays, more and more security-relevant data are stored on computer systems; security-critical business processes are mapped to their digital pendants. This situation applies to various critical infrastructures requiring that different security requirements must be fulfilled. It demands a way to design and express higher-level security policies for such critical organizations. In this paper we focus on authorisation policies to demonstrate how software engineering techniques can help validate authorisation constraints and enforce access control policies. Our approach leverages features and functionalities of the UML/OCL modeling methods as well as model driven approach to represent and specify authorisation model and constraints. Using our authorisation constraints editor, we articulate role-based authorisation policies. Also, we attempt to validate and enforce such constraints with the USE (UML Specification Environment) tool.

1. INTRODUCTION

Today information technology (IT) pervades more and more all aspects of our daily life. This applies to very different domains such as healthcare and digital government. On the other hand, new technologies go along with new risks, which must be systematically dealt with, such as preventing unauthorised access. Hence it is mandatory to establish adequate mechanisms that enforce the security and protection requirements demanded by the rules and laws relevant to the organisation in question.

Implementing such higher-level organisational authorisation policies in computer systems can be cumbersome and inefficient. However, it has turned out that one of the great advantages of role-based access control (RBAC) is that separation of duty (SoD) rules can be implemented in a natural way [4]. Generally speaking, role-based authorisation constraints are an important means for laying out higher-level access control policies [1, 5].

As demonstrated in [2, 7], the Unified Modeling Language (UML) and the Object Constraint Language (OCL) can be conveniently used to specify several classes of role-based authorisation constraints. Moreover, owing to the fact that OCL has proved its applicability in several industrial applications[1], OCL is a good means for such a practically relevant process like the design of access control policies.

However, in order to have a broader practical use, tool support is needed such that the specified authorisation constraints can be *automatically* enforced by the IT systems. Such an authorisation editor should be designed and implemented by sound software engineering techniques. In particular, the main focus should lie in the modeling process, whereas the implementation should be carried out routinely and as much as possible automatically. Hence, we demonstrate in this paper how to employ the USE system (UML Specification Environment) [8] to validate role-based security policies and to implement an authorisation tool. In particular, USE is a validation tool for UML models and OCL constraints, which has been reportedly applied in industry and research [8]. Since we employ UML/OCL for the specification of RBAC system, new software engineering techniques like Model Driven Devlopment (MDD) could be used for the development process of the authorisation editor.

The paper is now organised as follows: Section 2 gives a short overview of RBAC, UML/OCL, and MDD, and introduces the USE system. In Section 3 authorisation constraints are specified in OCL. Furthermore, it is described how the USE system can be employed to validate role-based authorisation policies. In Section 4 an authorisation editor is presented, which can enforce various kinds of authorisation constraints and which is based upon the USE system. Section 5 summarises and gives an outlook on future work.

*This work of Gail-J. Ahn was partially supported at the Laboratory of Information of Integration, Security and Privacy at the University of North Carolina at Charlotte by the grants from National Science Foundation (NSF-IIS-0242393) and Department of Energy Early Career Principal Investigator Award (DE-FG02-03ER25565)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Software Engineering for Secure Systems – Building Trustworthy Applications (SESS'05) St. Louis, Missouri, USA
Copyright 2005 ACM 1-59593-114-7/05/05 ...$5.00.

[1]OCL is UML's constraint specification language and UML has been widely adopted in software engineering discipline.

2. RELATED TECHNOLOGIES

We first give a short overview of RBAC, then we briefly describe UML/OCL and MDD, and introduce the USE tool, which can be employed to validate OCL constraints.

2.1 RBAC and Authorisation Constraints

RBAC has received considerable attention as an alternative to traditional discretionary and mandatory access control. One reason for this increasing interest is that in practice permissions are assigned to users according to their roles/functions in the organisation. In addition, the explicit representation of roles greatly simplifies the security management and allows to use well-known security principles like separation of duty and least privilege.

In the following, we briefly describe RBAC96, a family of RBAC models introduced by Sandhu et al. [10]. RBAC96 has the following components:

- Users, Roles, P, S (sets of users, roles, permissions, activated sessions)
- $UA \subseteq Users \times Roles$ (user assignment)
- $PA \subseteq Roles \times P$ (permission assignment)
- $RH \subseteq Roles \times Roles$ is a partial order also called the role hierarchy or role dominance relation written as $\leq$.

Users may activate a subset of the roles they are assigned to in a *session*. P is the set of ordered pairs of operations and objects. In the context of security and access control all resources accessible in an IT-system (e.g., files, database tables) are referred to by the notion *object*. An *operation* is an active process applicable to objects (e.g., read, write, append). The relation PA assigns each role a subset of P. So PA determines the operation(s) that each role may execute and the object(s) to which the operation in question is applicable for the given role.

An important advanced aspect of RBAC are *authorisation constraints*. Authorisation constraints are sometimes argued to be the principal motivation behind the introduction of RBAC [10]. They allow a policy designer to express higher-level organisational authorisation policies. Depending on the organisation, different kinds of authorisation constraints are required such as SoD in the banking field or constraints on delegation and context constraints in the healthcare domain [12].

2.2 Overview of UML and OCL

2.2.1 Unified Modeling Language.

The Unified Modeling Language (UML) [9] is a general-purpose visual modeling language in which we can specify, visualize, and document the components of software systems. It captures decisions and understanding about systems that must be constructed. UML has become a standard modeling language in the field of software engineering.

UML permits to describe static, functional, and dynamic models. In this paper, we concentrate on the static aspects of UML. A static model provides a structural view of information in a system. Classes are defined in terms of their attributes and relationships. The relationships include specifically associations between classes. In Figure 1, the conceptual static model for RBAC is depicted.

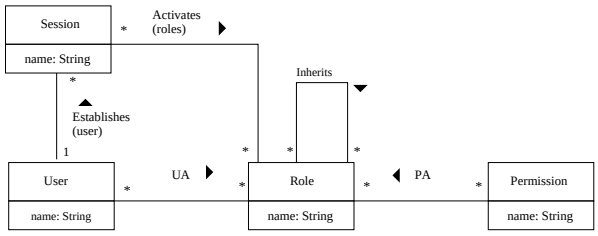

Figure 1: Conceptual Class Model for RBAC-Entity Classes.

2.2.2 Object Constraint Language.

The Object Constraint Language (OCL) [13] is a declarative language that describes constraints on object-oriented models. A constraint is a restriction on one or more values of an object-oriented model. OCL is an industrial standard for object-oriented analysis and design.

Each OCL expression is written in the context of a specific class. In an OCL expression, the reserved word `self` is used to refer to a contextual instance. The type of the context instance of an OCL expression is written with the `context` keyword, followed by the name of the type. The label `inv:` declares the constraint to be an invariant. Consider the RBAC model from Figure 1: If the context is `Role`, then `self` refers to an instance of `Role`. The following lines show an example of an OCL constraint expression describing a role with at most two users:

```
context Role inv: self.user->size()<2
```

`self.user` is a set of `User` objects that is selected by navigating from objects of class `Role` to `User` objects through an association. The ‘‘`.`’’ stands for a navigation. A property of a set is accessed by an arrow ‘‘`->`’’ followed by the name of the property. A property of the set of users is expressed using the `size` operation in this example.

The following shows another example describing that a user can be assigned to a role `r2` only if she is already member of `r1`:

```
context User inv:
self.role_->includes('r2') implies self.role_->includes('r1')
```

The expression `self.role_->includes('r2')` means that `r2` is a member of the set of roles the user is assigned to. The `implies` connector is similar to logical implication.

Furthermore, OCL has several built-in operations that can iterate over the members of a collection (set, bag, ...) such as forAll, exists, iterate, any and select (cf. [13]).

2.3 Model Driven Development

This section gives only a brief introduction into the main ideas behind Model Driven Development (MDD). To obtain more information about MDD, the interested reader is referred to [14, 13]. Key to MDD is the importance of models in the software development process. Within MDD, the software development process is driven by the activity of modeling the software system. The MDD process consists of three steps:

1. build a model with a high level of abstraction. This model is independent of any implementation technology and is called *Platform Independent Model* (PIM),

2. transform the PIM into one or more models tailored towards the specification constructs available in the specific implementation technology. These models are called *Platform Specific Models* (PSM),

3. generate code from the PSMs.

Several tools exist that can automatically produce code from a PSM. However, the new idea behind MDD is that the transformation step from the PIM to the PSM can also be carried out automatically by tools.

As pointed out in [13], the OCL can be regarded as a key ingredient of MDD. For example, the PSMs can be specified in OCL. In addition, the OCL can also be used to define languages for the PIMs and to create transformation definitions between PIMs und PSMs.

2.4 The USE Tool

This section explains the functionality of the UML Specification Environment (USE) which allows to validate UML and OCL descriptions. USE is now available as a mature version and has achieved more and more functionality since it was first introduced. USE is the only OCL tool allowing interactive monitoring of OCL invariants and pre- and postconditions and the automatic generation of non-trivial system states. These system states or system snapshots consist of the current objects and links between those objects adhering to the UML model in question.

The central idea of the USE tool is to check for software quality criteria like correct functionality of UML descriptions already on the design level in an implementation-independent manner. This approach takes advantage of descriptive design level specifications by expressing properties shorter and in a more abstract way. Such properties are given by invariants and pre- and postconditions, and these are checked by the USE system against the test scenarios, i.e., object diagrams and operation calls given by sequence diagrams, which the developer provides. These abstract design level tests are expected to be also used later in the implementation phase.

The USE tool expects as an input a textual description of a model and its OCL constraints (for an example of such a description refer to Figure 2). Then syntax checks of this description are carried out, which verify a specification against the grammar of the specification language, basically a superset of OCL extended with language constructs for defining the structure of the model. Having passed all these checks, the model can be displayed by the graphical user interface provided by the USE system. In particular, USE makes available a project browser which displays all the classes, associations, invariants, and pre- and post-conditions of the current model.

USE can then be employed to generate and change system states. A system state can be changed by issuing commands for creating and destroying objects, inserting and removing links between objects, and setting attribute values of objects. The USE system provides two different kinds of user interfaces for these commands, which can be used in parallel:

- a graphical interface supporting an intuitive approach

- a scripting interface providing a shell-like environment for experts allowing to define scripts with state manipulation commands like `!create e1:Employee`

The current system state can then be visualised in an object diagram window, which is automatically updated when the system state is changed.

The developer can check system states at any time. A system check includes two phases. First, all model-inherent must be verified. A model-inherent constraint is a constraint which is inherent to the semantics of all UML models. For example, the set of links between objects is verified against the multiplicity specifications of the association ends. Second, if the developer has defined explicit OCL constraints (invariants and pre- or post-conditions of operations), all the expressions are evaluated. If any constraint is false, the system state is considered ill-formed.

Moreover, the developer gets feedback from USE about the validity of the invariants in a special invariant window and the validity of the pre- and post-conditions in a sequence diagram window. Further information about the validity of invariants can be requested from USE by a dialog window for evaluating arbitrary OCL expressions. This dialog allows ad-hoc queries useful for navigating and exploring a system state at any time. Hence, USE helps the developer in analysing situations when an invariant fails. In order to give an impression how the work with USE looks like, Figure 3 depicts a USE screenshot with an example.

3. SPECIFICATION AND VALIDATION

In this section, we demonstrate how authorisation constraints can be specified in UML/OCL. Thereafter, it is described how the USE tool can be employed to validate role-based authorisation policies.

3.1 Constraints Specification

Subsequently, we give three examples that demonstrate how to use OCL to specify authorisation constraints.

Example 1: Simple Static Separation of Duty (SSOD)
The first example concerns a separation of duty constraint. Consider two (or more) conflicting roles such as accounts payable manager and purchasing manager. Mutual exclusion in terms of UA specifies that one individual cannot have both roles. This constraint on UA can be specified using the OCL expression as follows [2]:

```
context User inv SSOD:
let
   CR:Set={{AccountsPayableManager, PurchasingManager}, ...}
in
   CR->forAll(cr|cr->intersection(self.role_)->size()<=1)
```

Note that `CR` denotes here a set which consists of conflicting role sets, as introduced in [1].

Example 2: Prerequisite Roles
The second example is based upon the concept of prerequisite constraints as introduced in [10]. In this example, we consider a prerequisite constraint stating that a user can be assigned to the engineer role only if the user is already assigned to the employee role.

[2]For the sake of simplicity, we have left out here the part for the creation of the instances `AccountsPayableManager` and `PurchasingManager`. Similar remarks hold for the subsequent OCL specifications.

```
context User inv Prerequisite Role:
self.role_ ->includes(engineer) implies
self.role_->includes(employee)
```

Example 3: Static Separation of Duty - Conflict Users
By means of OCL even more complex authorisation constraints can be formulated. One example of such a constraint is SSOD-CU identified by Ahn in [1]. SSOD-CU (Static Separation of Duty - Conflict Users) means that two or more colluding users cannot be assigned to conflicting roles. For example, it might be the company policy that members of the same family cannot be assigned to the roles accounts payable manager and purchasing manager. SSOD-CU can now be expressed in OCL in the following way:

```
context User inv SSOD-CU:
let
   CU:Set(Set(User))=Set{Set{Frank,Susan},Set{Lars,Maria}},
   CR:Set(Set(Role))=Set{Set{Cashier,CashierSupervisor},
   Set{AccountsPayableManager,BillingClerk}}
in
   CR->forAll(cr|cr->intersection(self.role_)->size()<=1)
and
   CU->forAll(cu|
     CR->forAll(cr|cr->iterate(r:Roles;
     result:Set(User)=Set{}|
     result->union(r.user))->intersection(cu)->size()<=1))
```

Obviously, SSOD-CU is a composite constraint consisting of two parts, *an SSOD part* and *an additional part concerning the conflicting users*. The SSOD part is required because otherwise obviously the whole constraint would not be useful. The `iterate` operation iterates over all roles `r` belonging to a set of conflicting roles and collects all users of these roles. `CR` has the same meaning as in Example 1 whereas `CU` is a set consisting of all conflicting user sets.

3.2 Validation of RBAC policies

As mentioned in Section 2.4, the main application of the USE tool is the validation of UML/OCL models. The same can be carried out with a role-based authorisation policy. The USE specification of such an example policy is given in Figure 2 with the authorisation constraints expressed by OCL Constraints.

The aim of the validation of RBAC policies includes detection of conflicting constraints and identification of missing constraints. The validation can be done *before the deployment* of the RBAC policy, i.e., during the design phase. As pointed out above, the USE approach for validation is to generate system states and check these states against the specified constraints. In case of the role-based authorisation policy, this means we could create certain RBAC configurations and check if the RBAC configuration adheres to the access control policy. The RBAC configurations could be created automatically by running a script with the state manipulation commands, which are supported by the USE tool, or as an alternative the graphical user interface of USE can be used.

The result of the validation can lead to different consequences. Firstly, we may have reasonable system states that do not satisfy one or more authorisation constraints of our policy. This may indicate that the constraints are too strong or the model is not adequate. Secondly, the access control policy may allow undesired system states, i.e., the constraints are too weak. In the following both situations are discussed. This will be subsequently demonstrated by

```
model RBAC
--classes

class Roles
attributes
roleName:String
end

class Users
attributes
userName:String
end

class Permissions
attributes
op:Operations
o:Objects
end

class Objects
attributes
objectName:String
end

class Operations
attributes
operationName:String
end

class Sessions
attributes
sessionName:String
end

-- associations

association UA between
Users[*] role users
Roles[*] role roles
end

association PA between
Permission[*] role permission
Role[*] role role_
end

association establishes between
Users[1] role user
Session[*] role session
end

association activates between
Session[*] role session
Role[*] role role_
end

association inherits between
Role[*] role senior
Role[*] role junior
end

constraints

context Users inv PrerequisiteRole:
self.role_->includes(r2)
implies self.role_->includes(r1)

context Role inv SSOD-CU:
let
CU:Set(Set(User))=Set{{u1,u2,u3},{u4,u5}}
in
let
CR:Set(Set(Role))=Set{Set{r1,r2},...}
in
CU->forAll(cu|
CR->forAll(cr|cr->iterate(r:Role;
result:Set(User)=oclEmpty(Set(User))|
result->union(r.user))->
   intersection(cu)->size()<=1))
```

Figure 2: USE specification of a role-based authorisation policy.

an example, considering the RBAC policy presented in Figure 2. Clearly, this example policy is rather simple, but in reality we often have to deal with considerably more complex policies. Now, let us further assume that the policy designer has forgotten that he had once defined a prerequisite role constraint between $r1$ and $r2$. Later, the policy designer decided to define $r1$ and $r2$ mutually exclusive due to a change of organisational rules/policies. Obviously, both constraints could not be satisfied at the same time and hence the composite constraint is too strong. The USE screenshot in Figure 3 displays the situation after user u has been assigned to $r2$. Clearly, the policy designer cannot have assigned u to role $r1$; otherwise the new SSOD constraint would be violated. However, now the constraint `User::PrerequisiteRole` is evaluated to false (cf. "Class invariants" view in Figure 3), and hence the current RBAC configuration is not a correct system state according to the given policy specification.

Admittedly, the mere information that a constraint is false might often not help to find the real reason for the problem and to resolve the conflict. Additional information is required which objects and links of the current state violate the constraint. For such a purpose, the policy designer can debug the constraints that are not satisfied by the current system state with the "Evaluate OCL expression" dialog made available by USE. For example, in Figure 3 the result of the query "all users who are assigned to $r2$ but not to $r1$"

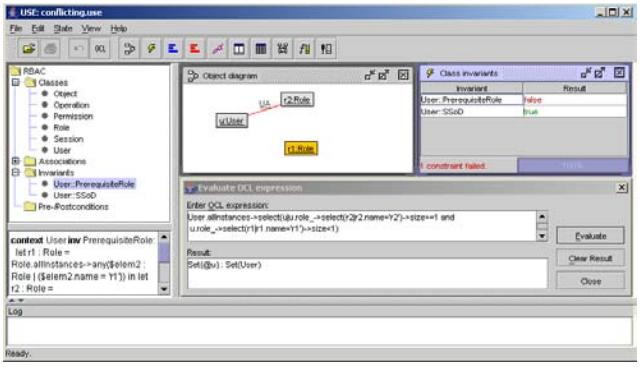

Figure 3: USE screenshot: two conflicting constraints.

applied to the given RBAC configuration is shown. Here, one can learn that u is not assigned to $r1$, although this is required by the prerequisite role constraint. If the policy designer now conversely tries to assign u to $r1$, the SSOD constraint fails, and as a consequence one can conclude that both constraints are contradictory. A policy designer could now employ USE in a similar way for other constraint types such as cardinality constraints or other SoD properties. In particular, this approach is helpful if a new constraint is added to the policy, in order to check if it is in conflict with the composition of the already defined constraints.

3.2.1 Detection of Missing Constraints

The second consequence of the constraint validation may be that the policy permits undesirable system states, i.e., the authorisation constraints are too weak. Once again suppose that the policy designer has defined a complex access control policy. Let us further assume that she has missed to define the SSOD part of the SSOD-CU constraint mentioned above and that a system state has been created by USE in which u is assigned to both the roles $r1$ and $r2$. Now, USE can help detecting the missing constraint in this scenario: all constraints (in our case specifically the conflict user part of the SSOD-CU constraint) defined so far are evaluated to true and hence the policy seems supposedly to be correct. On the other hand, the policy – in particular, even the combination of authorisation constraints – permits a user being assigned to the mutually exclusive roles $r1$ and $r2$. Therefore, a further SSOD constraint must be added to the policy in order to exclude the undesired state and to obtain a more restrictive access control policy. This means that the constraints must be strengthened.

4. AUTHORISATION EDITOR

In this section, it will be demonstrated how an RBAC authorisation editor could be built based upon the functionality of the USE system. This tool can enforce several kinds of authorisation constraints like those listed in [1]. More explicitly speaking, the authorisation editor can be used in principle to specify and enforce all authorisation constraints expressible in OCL. As a consequence, types of authorisation constraints beyond those enumerated in [1] can also be formulated and enforced.

In the following, the functionality of the authorisation editor will be presented. Thereafter, it will be described more thoroughly how this tool has been implemented.

4.1 Functionality of the Authorisation Editor

First, the prototype of the authorisation editor currently supports most of the functionality demanded by the RBAC standard [3]. This means that we have implemented

- administrative functions,
- system functions, and
- review functions.

According to [3] *administrative functions* allow for the creation and maintenance of the element sets (e.g., User, Role, Permission) and relations (e.g., UA, PA, RH) of the RBAC models. For example, $AddUser$, and $AssignUser$ belong to this class of functions. *System functions* are required by the RBAC authorisation editor for session management and making access control decisions. Thus, examples are *CreateSession* and *CheckAccess*. The third class of functions supported by the authorisation editor are *review functions*, which allow for reviewing the results of the actions created by administrative functions. Typical examples of review functions are $AssignedUsers$ and $UserPermissions$.

Beyond this basic functionality, the RBAC authorisation editor provides mechanisms for defining and enforcing both role hierarchies and authorisation constraints. Currently, the following types of authorisation constraints can be enforced with the help of this tool:

- various kinds of SoD properties (such as simple static SoD, simple dynamic SoD, object-based static SoD),
- various kinds of cardinality constraints,
- prerequisite roles and permissions.

The tool is by no means constrained to the aforementioned authorisation constraints. Other types of constraints such as mechanisms for context roles, constraints on the delegation process [12], or the SSOD-CU constraint could also be implemented. In fact, the tool can be quite easily extended to support other authorisation constraints such that it is flexible enough to enforce various role-based authorisation policies, depending on the internal rules of the organisation in question.

The authorisation editor can also deal with role hierarchies, which are not restricted to inheritance trees, but can also in general form directed acyclic graphs. Moreover, the tool can detect and then prevent inconsistencies such as a senior role which inherits two mutually exclusive junior roles.

To give a better overview, a screen shot of the current prototype of the authorisation editor is shown in Figure 4. In the upper part of the window, there are several buttons, each button stands for a special administrative functions. The large window in the middle of the tool visualises the current system state (RBAC configuration). The visualisation of the system state will be immediately renewed when the system state has been changed by an administrative function. At the bottom of the window there is a log window, which displays the result of the last applied administrative function, i.e., a confirmation that the last operation has succeeded, or a short text if an error occurred. There are currently two windows open: On the right-hand side there is a small

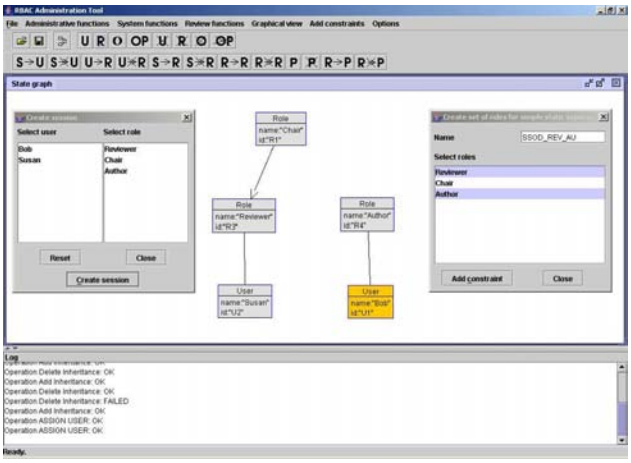

Figure 4: The authorisation editor.

window to create a set of roles for a simple static SoD constraint; on the left-hand side there is a window to create a session for a user with active roles.

4.2 Implementation Aspects

As pointed out above, the authorisation editor has been implemented by using an API made available by the USE tool. This way, the functionality of USE is hidden from the administrator/security officer by the graphical user interface of the authorisation editor.

Now our implementation, based upon the USE API, will be described in more detail. In particular, we explain how the administrative functions, system functions, and review functions have been realised. Thereafter, the constraint checking mechanism is sketched and it is discussed how the development process chosen for the implementation of the authorisation editor fits into the MDD framework.

4.2.1 Administrative Functions

The core operations provided by an RBAC authorisation editor are administrative functions. With these functions, an administrator can change the RBAC configuration. Administrative functions can be easily implemented by using state manipulation commands made available by the USE system. To demonstrate this, we subsequently consider the operation *AssignUser* which assigns a user to a role. The *AssignUser* operaration can be expressed by the following state manipulation command of the USE system:

`!insert (u,r) into UA` (with a user u and a role r).

This command could then be called by employing the command execution facility provided by the USE API called `executeCmd()`. Furthermore, the other administrative functions can be realised in a similar way. Clearly, in order to remain in a consistent state, all (relevant) authorisation constraints must also be checked. This will be explained in more detail below.

4.2.2 Review Functions

RBAC review functions are demanded by the RBAC standard and can also be conveniently implemented employing the USE functionality. For this purpose, the query facilities of USE can be employed (cf. Section 3.2). In particular, the USE API provides the method `eval` which evaluates a query consisting of an OCL expression in the current system state. For example, the following OCL query expresses the *UserPermissions* function, which returns all permissions belonging to a user:

```
UserPermissions(u:User):Set(Permission)=
u.role_->iterate(r:Role;
     result:Set(Permission)={}|
     result->union(r.permission))
```

This query can be specified in a USE file (such as that presented in Figure 2) which can be read when the authorisation editor is started. Then the `eval` method of the expression evaluator provided by the USE system can be invoked with the two parameters "UserPermissions" and the user u, whose set of permissions is to be determined. The other review functions can be implemented similarly.

4.2.3 System Functions

System functions can be realised with USE similarly to the review functions. This will be demonstrated by the *CheckAccess* function, which decides if a session s is permitted to execute operation *op* on object o. Hence, we must define an appropriate OCL expression:

```
CheckAccess(op:Operation,o:Object,s:Session):Boolean=
let ps=s.role_->iterate(r:Role;
     result:Set(Permission)={}|
     result->union(r.permission))
in
     ps->exists(p|p.o=o and p.op=op)
```

As in the case of review functions, the *CheckAccess* function can be specified in the USE file and can then be executed by the aforementioned `eval` method with the correct actual parameters for *op*, o, and s. In contrast, the session-related system functions like *CreateSession* must be realised in the same way as the administrative functions by means of the state manipulation commands.

4.2.4 Constraint Checking

The basic idea of the constraint checking mechanism is according to [6] as follows: The authorisation editor, or to put it in another way, the USE system checks if the relevant authorisation constraints are still satisfied *after* an administrative or system function has been carried out. This is done by the `check` method made available by the USE API. If any constraint is violated, the last administrative or system function is automatically revoked with the help of an `undo` method. As a consequence, the tool produces only states that are consistent with the specified authorisation constraints.

Templates for the different types of authorisation constraints can be read directly from the USE specification file as OCL invariants during the start of the authorisation editor. The OCL specifications for the authorisation constraints are slightly different from those given in Section 3 for implementation reasons [6]. In particular, special UML classes for the sets of conflicting roles or conflicting users have been introduced. This allows for setting parameters (e.g., conflicting roles) at runtime.

4.2.5 Aspects of Model Driven Development

As pointed out in the previous section, templates for the different types of authorisation constraints can be specified

in the USE file. Hence, the authorisation editor can be easily extended with new types of authorisation constraints such as context roles [12]. Due to the fact that a growing number of applications process security-critical data, it is expected that several new types of autorisation constraints might be identified in the future, depending on the protection requirements of the applications in question. For this reason, the ability to extend the authorisation editor with new classes of constraints is a crucial aspect of the development process described in this paper.

Specifically, the main work for the realisation of new constraint types lies in the modeling or specification process. Clearly, some modifications on the authorisations editor's graphical user interface must be carried out to support the new constraint types. In addition, the functions of the USE API which check the authorisations constraints are called manually such that we have currently only a semi-automatic code generation process. Nevertheless, this form of semi-automatic code generation can be seen as a first step towards MDD on developing the authorisation editor.

Therefore, we now briefly discuss the relationship between the development process we employed for the authorisation editor with MDD. In our case, the USE specification of the RBAC model can be regarded as the PIM (Platform Independent Model) while the concrete USE API calls of the implementation can be seen as the PSM (Platform Specific Model). Strictly speaking, however, graphical elements like dialog windows and menue items are still missing in this PSM. Moreover, an automatic, hopefully tool-supported, transformation process between the PIM and PSM would also be required. At this moment, however, we pursue only the goal to explain how our development of the authorisation editor fits into the MDD framework. Due to the fact that we used UML/OCL for the specification of constraints, it should be possible to build a tool which helps in automatically transforming the UML/OCL specifications of the RBAC specification into an authorisation engine. This remains future work.

5. CONCLUSION AND FUTURE WORK

In this paper we demonstrated that with the help of OCL several classes of authorisation constraints can be specified. Due to the fact that the UML/OCL is quite familiar in industrial environments there is hope that OCL can be used by policy designers in different organisations. In addition, we have demonstrated how the USE tool, a validation tool for OCL constraints, can be employed to validate authorisation constraints against RBAC configurations. Specifically, we showed the USE-based approach could help a policy designer to detect certain conflicts between authorisation constraints and find missing constraints.

Moreover, the USE system can not only be employed to validate an access control policy during the design phase. but also be helpful to implement an RBAC authorisation editor. This tool provides administrative, system, and review functions as demanded by the RBAC standard [3]. Moreover, the tool supports role hierarchies and can enforce several types of authorisation constraints. In addition, we argued that the tool can be easily extended with new types of authorisation constraints.

Due to the fact that USE can only check the current snapshot, history-based authorisation constraints [11] cannot be dealt with. For this purpose a temporal extension of OCL like that introduced in [15] is needed. Hence, it remains future work to extend USE and the authorisation tool in order to deal with history-based constraints. In the long run, it is envisioned to develop an authorisation tool automatically from an RBAC specification. For this task, appropriate MDD transformation tools should be developed.

6. REFERENCES

[1] G.-J. Ahn, *The RCL 2000 language for specifying role-based authorization constraints*, Ph.D. thesis, George Mason University, Fairfax, Virginia, 1999.

[2] G.-J. Ahn and M.E. Shin, *Role-Based Authorization Constraints Specification Using Object Constraint Language*, Proc. of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise, IEEE, 2001, pp. 157–162.

[3] American National Standards Institute Inc., *Role Based Access Control*, 2004, ANSI-INCITS 359-2004.

[4] D.F. Ferraiolo, D.R. Kuhn, and R. Chandramouli, *Role-based access control*, Artec House, Boston, 2003.

[5] T. Jaeger and J.E. Tidswell, *Practical safety in flexible access control models*, ACM TISSEC **4** (2001), no. 2, 158–190.

[6] L. Migge, *Specification and Enforcement of Role-based Security Policies*, 2005, Master Thesis, Universität Bremen.

[7] I. Ray, N. Li, R. France, and D.-K. Kim, *Using UML to visualize role-based access control constraints*, Proc. of the 9th ACM symposium on Access control models and technologies, ACM Press New York, USA, 2004, pp. 115–124.

[8] M. Richters, *A Precise Approach to Validating UML Models and OCL Constraints*, Ph.D. thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[9] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual, Second Edition*, Object Technology Series, Addison Wesley Longman, Reading, Mass., 2004.

[10] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman, *Role-based access control models*, Computer **29** (1996), no. 2, 38–47.

[11] R. Simon and M. Zurko, *Separation of duty in role-based environments*, 10th IEEE Computer Security Foundations Workshop (CSFW '97), June 1997, pp. 183–194.

[12] K. Sohr, M. Drouineaud, and G.-J. Ahn, *Formal Specification of Role-based Security Policies for Clinical Information Systems, Santa Fe, New Mexico*, Proc. of the 20th ACM Symposium on Applied Computing, 2005, To appear.

[13] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting your models ready for MDA*, Addison-Wesley, Reading/MA, 2003.

[14] J. Warmer, A. Kleppe, and W. Bast, *The MDA explained – the model driven architecture: Practice and promise*, Addison-Wesley, Reading/MA, 2003.

[15] P. Ziemann and M. Gogolla, *An OCL Extension for Formulating Temporal Constraints*, Research Report 1/03, Universität Bremen, 2003.