# Specification and Validation of Authorisation Constraints Using UML and OCL

Karsten Sohr[1], Gail-Joon Ahn[2], Martin Gogolla[1], and Lars Migge[1]

[1] Department of Mathematics and Computer Science
Universität Bremen
Bibliothekstr. 1
28359 Bremen, Germany
[2] Department of Software and Information Systems,
University of North Carolina at Charlotte
Charlotte, NC 28223, USA

**Abstract.** Today 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 domains such as healthcare industry, digital government, and financial service institutes requiring that different security requirements must be fulfilled. Authorisation constraints can help the policy architect design and express higher-level security policies for such organisations. Although the importance of constraints has been addressed in the literature, there does not exist a systematic way to validate and test authorisation constraints. In this paper, we attempt to specify non-temporal constraints and history-based constraints in Object Constraint Language (OCL) which is a constraint specification language of Unified Modeling Language (UML) and describe how we can facilitate the USE tool to validate and test such policies. We also discuss the issues of identification of conflicting constraints and missing constraints.

## 1 Introduction

Today information technology pervades more and more our daily life. This applies to very different domains such as healthcare, e-government, banking. 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. For example, in Europe there do exist strong data protection requirements as those formulated in the Directive 95/46/EC [8]. This directive among other areas applies to clinical information systems where in particular the principle of patient consent must be enforced [4]. In contrast, in the banking domain other

security requirements such as data integrity are more important such that often separation of duty policies (SoD) [19, 5] must be enforced.

Implementing such higher-level organisational security 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 SoD rules can be implemented in a natural way [10]. Generally speaking, role-based authorisation constraints are an important means for laying out higher-level security policies [1, 15]. Although there is several work on the specification of role-based authorisation constraints, e.g., [1, 15], there is a lack of appropriate tool support for the validation, enforcement, and testing of role-based access control policies. Specifically, tools are needed which can be applied quite easily by a policy designer without too much deeper training.

As demonstrated in [2, 20], 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 security policies.

However, as mentioned above, tool support is needed in order to have a broader practical use. Hence, we demonstrate in this paper how to employ the USE system (UML Specification Environment) [21, 22] to validate and test access control policies formulated in UML and OCL. In particular, USE is a validation tool for UML models and OCL constraints, which has been reportedly applied in industry and research [21]. With the help of this tool, a policy designer can detect conflicting and missing authorisation constraints. In addition, it will be shown in this paper how USE can be employed for testing a security policy after the deployment of authorisation constraints. This may even be helpful for an external policy reviewer, who can use the tool to check if a given configuration involving users, roles, permissions, etc. satisfies certain conditions, e.g., data protection requirements.

The paper is now organised as follows: Section 2 gives a short overview of RBAC, UML/OCL, and introduces the USE system. In Section 3 typical and partly more complex authorisation constraints are specified in OCL and in a temporal OCL extension. Section 4 then demonstrates how USE can be employed to validate and enforce RBAC security policies and

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

test RBAC configurations while Section 5 sketches related work. Section 6 summarises and gives an outlook on future work.

## 2 Related Technologies

We first give a short overview of RBAC, then we briefly describe UML and OCL, and finally 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 (governmental or commercial) [9]. 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. [24]. 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 to each role a subset of $P$. So $PA$ determines for each role the operation(s) it may execute and the object(s) to which the operation in question is applicable for the given role. Thus any user having assumed this role can apply an operation to an object if the corresponding ordered pair is an element of the subset assigned to the role by $PA$.

An important advanced aspect of RBAC are *authorisation constraints*. Authorisation constraints are sometimes argued to be the principal motivation behind the introduction of RBAC [24]. They allow a policy designer

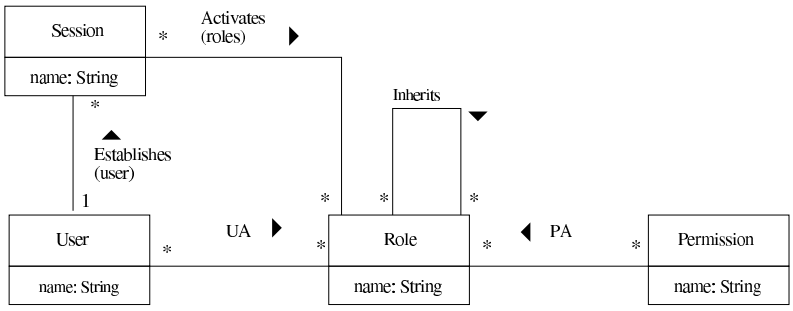


**Fig. 1.** Conceptual Class Model for RBAC-Entity Classes.

to express higher-level organisational security policies. Depending on the organisation, different kinds of authorisation constraints are required such as SoD in the banking field [5] or constraints on delegation and context constraints in the healthcare domain [26]. Later in this paper, different kinds of authorisation constraints are specified and discussed.

### 2.2 Overview of UML and OCL

**Unified Modeling Language.** The Unified Modeling Language (UML) [23] 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.

**Object Constraint Language.** The Object Constraint Language (OCL) [27] 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. [27]).

### 2.3 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.

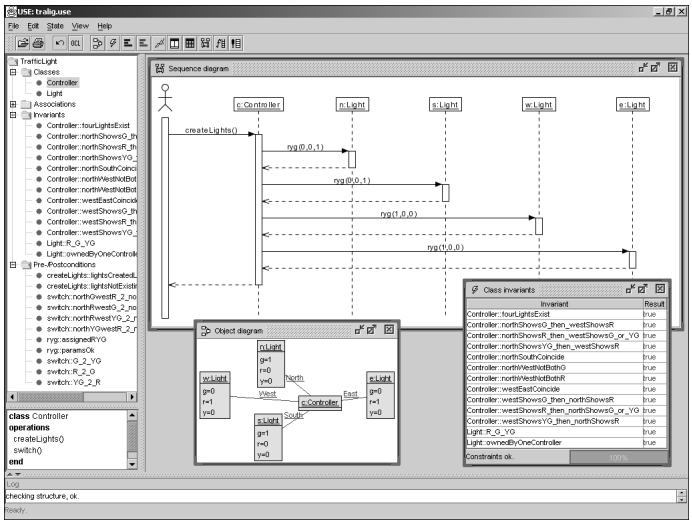


**Fig. 2.** USE screen shot.

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 3). 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.

Figure 2 shows a USE screen shot with an example. On the left we see the project browser displaying the classes, associations, invariants, and

operation pre- and post-conditions. In a detail window below, the selected class is pictured with all details. On the right, we identify a sequence diagram presenting the operations which lead to the current system state given in the object diagram window below. The evaluation of the invariants in this system state is pictured in the class invariant window to the right of the object diagram window. The developer gets feedback from USE about the validity of the invariants in the invariant window and the validity of the pre- and post-conditions in the sequence diagram window. Further information about the validity of invariants can be requested 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 or a pre- or post-condition fails. This query window will be used several times in Section 4.

## 3 Constraints Specification

In this section, different types of authorisation constraints are specified in OCL. In the first subsection, non-temporal authorisation constraints are formulated in OCL, whereas in the second subsection history-based authorisation constraints are formalised in a temporal extension of OCL.

### 3.1 Non-temporal Authorisation Constraints

Subsequently we give four 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)
```

[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.

This formulation of SSOD is based upon the the SSOD specification given in [1]. Specifically, `CR` denotes a set which consists of conflicting role sets.

*Example 2: Prerequisite Roles*
The second example is based upon the concept of prerequisite constraints as introduced in [24]. 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.

```
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{Michael,Frank,Susan},Set{Lars,Maria}},
   CR:Set(Set(Role))=Set{Set{AccountsPayableManager, BillingClerk},
   Set{Cashier, CashierSupervisor}, ...}
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))
```

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.

```
model RBAC
-- classes

class Role
attributes
   name:String
end

class User
attributes
   name:String
end

class Permission
attributes
   name:String
   op:Operation
   o:Object
end

class Object
attributes
   name:String
end

class Operation
attributes
   name:String
end

class Session
attributes
   name:String
end

-- associations
association UA between
   User[*] role user
   Role[*] role role_
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)

--constraint: user part of SSOD-CU
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))
```

**Fig. 3.** USE specification of an RBAC security policy.

*Example 4: Role Activation Process of Context Roles*
In the last example the permission activation mechanism for context roles as described in [28] is formalised in OCL. Context roles may assume different versions depending on the context where the roles are activated (e.g., location, time).

In order to apply context constraints, the traditional permission activation process is changed as follows. After successful authentication, the user selects a subset of roles for being activated in the current session. The anchored context constraints from the user (e.g., location) are retrieved and applied at role activation; after successful role activation, the anchored context constraints are further applied to the permissions of the activated roles; this process can be formulated in OCL as follows:

```
context User inv ContextConstraint:
self.role_->forAll(r|r.permission->forAll(p|p.o.location<>'' implies
```

```
(p.o.location=self.location and
self.session->exists(s|s.role_->includes(r)))))
```

We assume here that both the user and the object have an additional attribute that describes the current location of the user and the object, respectively. Furthermore, `p.o.location<>''` indicates that the object has a location attribute.

Given the expressive power of OCL, at least the constraints expressible in the authorisation constraint specification language RCL 2000 [1] can be formulated in OCL. In particular, the `OE` function provided by RCL 2000 can be translated into OCL's `any` operation, e.g., `OE(User)` is translated into `User.allInstances->any(true)`. RCL 2000's `AO` function can then be expressed using the `excluding` operation of OCL.

### 3.2 History-based constraints

OCL is quite similar to first-order predicate logic. As expressions of the predicate calculus, OCL expressions used in invariants are evaluated in a system state. However, due to the fact that we consider here only one snapshot of the system, we have no notion of time. Hence, authorisation constraints that consider the execution history such as history-based or object-based dynamic SoD [12] cannot be adequately expressed.

In the following, we sketch how history-based authorisation constraints can be specified in TOCL (Temporal OCL) [29], an extension of OCL with temporal elements. In particular, temporal operators like `always` (in the future), `sometime` (in the future), and `next` are available.

To demonstrate how history-based authorisation constraints can be formulated in TOCL, we take dynamic object-based SoD as an example, which has been introduced informally by Nash and Poland [19]. Dynamic object-based SoD roughly speaking means that a user must not act upon an object that the same user has previously acted upon. Other dynamic SoD constraints enumerated in [12] can clearly be expressed in TOCL, too.

In order to specify dynamic object-based SoD in TOCL, we use two predicates introduced in [18], namely $auth(u, op, obj)$ and $exec(u, op, obj)$. The former predicate means that a user $u$ is authorised to execute operation $op$ on object $obj$ while the latter means that user $u$ executes operation $op$ on object $obj$ in the present state. For the sake of simplicity, the full details of those predicates are left out here. The interested reader is referred to [18] to obtain more information on that topic.

Due to the fact that `exec` and `auth` are ternary predicates and OCL supports only binary associations we extend (T)OCL with additional predicates `Exec` and `Auth` to express ternary associations, as proposed in [14].

With this extension, we obtain the following TOCL specification for object-based dynamic SoD (using the `always` operator):

```
context Object inv ObjDSoD:
Operation.allInstances->forAll(op,op1|
   User.allInstances->forAll(u|
     (Exec(u,op,self) and op1<>op) implies always not Auth(u,op1,self))))
```

This corresponds to the specification of dynamic object-based SoD in first-order linear temporal logic as given in [18]:
$\forall u : Users; op, op1 : OpSet; obj : Object.op \neq op1 \wedge exec(u, op, obj)$
$\Rightarrow \Box\neg auth(u, op1, obj)$

## 4 Validation and Testing of RBAC security policies

With OCL we have a light-weight formalism at hand, which can help specifying RBAC security policies. What is however missing, is a tool which helps a policy designer in validating her RBAC policy. Hence, in the sequel it will be demonstrated how the USE tool can be employed for this purpose (cf. Section 4.1). Specifically, authorisation constraints such as those categorised in [1] can be handled. Additionally, USE can also be applied to test concrete RBAC configurations against certain conditions; this point is presented in Section 4.2. The last section sketches how the USE functionality can be used to build an RBAC authorisation editor.

### 4.1 Validation of RBAC security policies

As mentioned in section 2.3, the main application of the USE tool is the validation of UML models, which consist of classes, associations, and OCL constraints. The same can be carried out with an RBAC security policy. The USE specification of a security policy (UML/OCL model) is given in Figure 3 with the authorisation constraints expressed by OCL constraints. This policy will serve as an example within this section.

At this point, we should differentiate between administrators and policy designers. The latter are responsible for designing the policies whereas the former implement these policies. We make this distinction because a policy designer should possess significant knowledge about organisational rules. In addition, we expect that policy designers are more familiar with modeling languages and validation tools like USE than administrators.

Through the validation of RBAC policies conflicting constraints can be detected and missing constraints identified. The validation can be done *before the deployment* of the RBAC policy, i.e., during the design phase. As indicated above, the USE approach for validation is to generate system states (snapshots) and check these states against the specified constraints. In our case, the system states are certain RBAC configurations (consisting of users, roles, the relations between these entities). 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 with a graphical user interface provided by the USE system. This animation-based approach for the validation of security policies can be regarded as a complement to a rigorous formal verification, which often requires deeper training in formal methods.

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 the policy. This may indicate that the constraints are too strong or the model is not adequate. Secondly, the security policy may allow undesired system states, i.e., the constraints are too weak. In the following both situations are discussed more thoroughly.

**Conflicting Constraints.** USE may help the policy designer find conflicting constraints. This will be subsequently demonstrated by an example, considering the RBAC policy presented in Figure 3. 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 4 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 4), 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

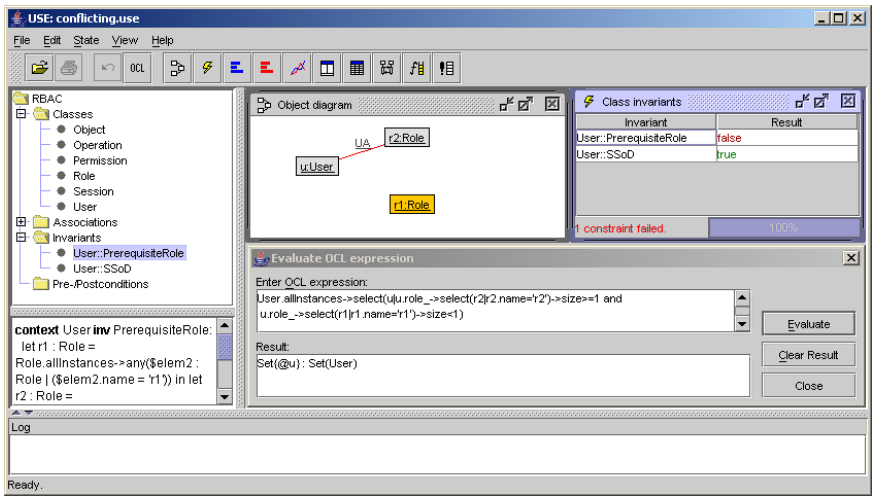


**Fig. 4.** USE screenshot: two conflicting constraints.

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 4 the result of the query "all users who are assigned to $r2$ but not to $r1$" 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, i.e., if at least one of these constraints is evaluated to false.

Nevertheless, USE may find conflicts only in certain cases, and there is no guarantee that all conflicts can be detected. Had $u$ not been assigned to $r2$, the conflict would have remained undetected. In order to eliminate contradictory constraints in general, a more formal approach such as theorem proving or model checking is required. On the other hand, the USE approach is only meant to improve the design of a security policy, but does not aim at a formally proven design. Given the condition that there is often a lack of tools for policy analysis, the USE approach can be considered as a first pratical step towards more reliable security mechanisms.

However, various heuristics can be applied which may streamline the conflict detection process with USE. For example, system states (snapshots) could be created which are *specially tailored* towards certain constraint types. In particular, we could consider snapshots which satisfy the authorisation constraint in question and which contain all the parameters (objects and links) occurring in this constraint (cf. the system state in Figure 4 for the SSOD constraint). Such a snapshot can then be taken as a *starting point* for the conflict detection process. Specifically, we can check if this system state adheres at the same time to the composition of the other already defined constraints. As a further improvement, we could store snapshot templates for each constraint type (e.g., SSOD, prerequisite roles) and instantiate these templates for a certain constraint if needed. This way, a library with snapshot templates is available, which can be reused and appropriately combined with other snapshots to obtain test cases for conflict detection.

**Detection of Missing Constraints.** The second consequence of 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 security policy. Let us further assume that she has missed to define the SSOD part of the SSOD-CU constraint mentioned above (cf. Figure 3) and that an undesirable system state has been created by USE in which $u$ is assigned to both the roles $r1$ and $r2$. Now, USE can help in 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 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 undesirable state and to obtain a more restrictive security policy.

But how can we create a system state which reveals the missing constraint? One possible solution is to create an RBAC configuration tailored towards the missing constraint as described in the previous section, but with the difference that now snapshots must be considered that violate the missing constraint. Another possibility is to use the test generator provided by USE [13]. By means of this generator we can create system states at random and then check if the created system state violates certain conditions with the help of the "Evaluate OCL expression" dialog. For example, we can check by the query "all users who are assigned to

$r2$ and to $r1$" if the SSOD constraint is violated. If the resulting set is non-empty, we have a system state that violates SSOD. If at the same time all authorisation constraints are true, one or more constraints are missing (cf. the remarks in the previous paragraph).

### 4.2 Testing a Given RBAC Configuration with USE

Beyond the validation of constraints, the aforementioned features of USE can also be employed for testing an RBAC configuration *after* the constraints/policy have been deployed. However, observe that we consider here a *predefined* configuration of users, roles, user assignments, etc. which corresponds to a real-world RBAC configuration of an organisation.

Testing an RBAC configuration may be mandatory in several situations. For example, in some domains (e.g., healthcare) strict data protection laws must be fulfilled such as the European Directive 95/46/EC [8]. In order to assess the current RBAC configuration defined for security-relevant applications, often some external review is required, e.g., from an government agency responsible for data protection as established in Germany. What is often missing, is a tool that supports an external reviewer in checking a concrete RBAC configuration of an organisation against certain properties such as data protection rules. In addition, the ability to test RBAC configurations may also be helpful for administrators in order to check if a security policy has been implemented correctly.

USE can now be employed as an ad hoc query tool to check certain properties (specifically, authorisation constraints) of the current RBAC configuration such as:

- there is no common user of mutually exclusive roles
- only clinicians of a patient's current ward may have access to the patient's electronic patient record[3]

For this purpose, the "Evaluate OCL expression" dialog is helpful again. For example, a reviewer can check the current RBAC configuration if and which users are assigned to the roles $r1$ and $r2$, which ought to be mutually exclusive. Due to the fact that an administrator or external reviewer usually is not an expert in specification formalisms like OCL an authorisation editor should be made available which hides the formalism behind a graphical user interface (GUI). This is discussed in the following.

[3] We assume here that there is a further attribute "ward" for certain roles and for users.

### 4.3 Authorisation Editor

We have implemented an RBAC authorisation editor built upon the Java API made available by the USE system. This way, the USE system is hidden from the administrator and hence she need not be familar with UML/OCL and USE. The authorisation editor can enforce several types 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 such as context constraints.

In the following, the functionality of the authorisation editor will be presented in more detail. First, the prototype of the authorisation editor currently supports most of the functionality demanded by the ANSI standard for RBAC [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 the RBAC relations (e.g., `UA`, `PA`). For example, `AddUser`, `DeleteUser`, and `AssignUser` belong to this class of functions. *System functions* are required by the authorisation editor for session management and making access control decisions. Thus, examples are `CreateSession`, and `AddActiveRole`. 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`. Administrative and system functions can be implemented by the state manipulation commands provided by the USE system. Due to the aforementioned query facilities of USE, RBAC review functions can also be easily implemented.

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 basic idea of the constraint checking mechanism is 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 function has been carried out. If any constraint is violated, the last function is automatically revoked. As a consequence, the tool can only produce states that are consistent with the specified constraints.

The tool is by no means constrained to the aforementioned authorisation constraints. Other types of constraints such as mechanisms for context roles [11], constraints on the delegation process [28], 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. For example, assume we have a role *Chair* and two junior roles *Reviewer* and *Author*. Further assume both junior roles are mutually exclusive. Then the role *Chair* is strictly speaking useless because no user can ever be assigned to this role. To give a better overview, a screen

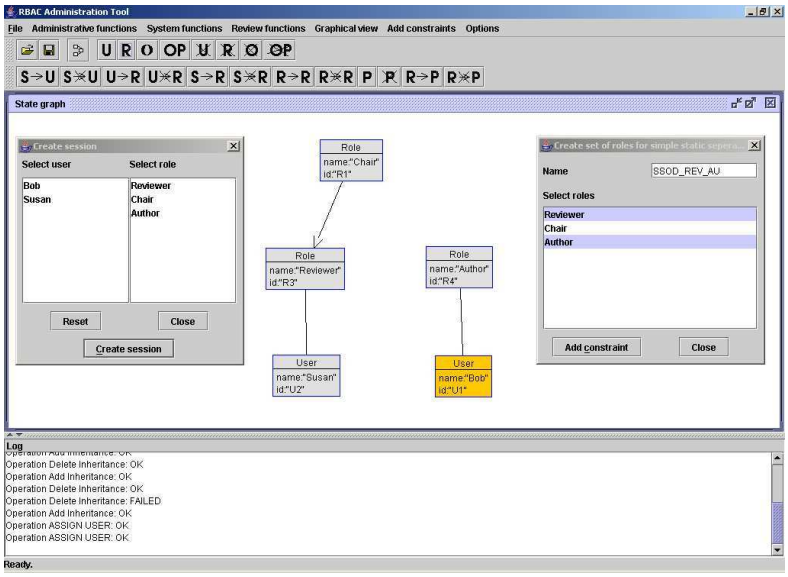


**Fig. 5.** The authorisation editor.

shot of the current prototype of the authorisation editor is shown in Figure 5. In the upper part of the window, there are several buttons, each

button stands for a special administrative function. 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 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.

## 5 Related Work

There is several work concerning the specification of RBAC authorisation constraints, e.g., a graphical language [15] and the RCL 2000 language based upon restricted first-order logic [1]. As demonstrated in [1], various classes of authorisation constraints – among them user-, permission-, object-based SoD constraints – can be expressed with RCL 2000. Although the classification and the case studies are insightful, no tool support for constraint validation, enforcement, and testing has been implemented so far. In [20] and [2], constraints are formulated in UML/OCL, but once again no tool support for the validation is available. In this respect, the USE approach fills this gap.

In [16, 17], another approach for the verification of RBAC policies is presented, which is based upon graph transformations. However, this approach does not tackle the problem of conflicting constraints, but the problem of graph rules conflicting with constraints. Due to the fact that several constraints can only be expressed clumsily (e.g., SSOD-CU, operational SoD) a formulation of those constraints in OCL is often more intuitively. Furthermore, performance problems do exist with the graph transformation approach because strictly speaking attributed graph grammars are required to label the nodes (e.g., roles, users) and the graph matching algorithm is in this case NP-hard [7].

In [6], an authorisation editor is presented which is similar to the one described in Section 4.3. However, with the approach from [6], for example, neither the SSOD-CU constraint nor context constraints can be specified and enforced. On the other hand, with USE no history-based SoD constraints can be enforced because TOCL is currently not supported.

## 6 Conclusion and Future Work

In this paper we demonstrated that with the help of OCL several classes of authorisation constraints and even complex composite 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 demonstrated how the USE tool, a validation tool for OCL constraints, can be employed to fulfill several practical needs. Firstly, USE can be employed to validate authorisation constraints against RBAC configurations (system states). Specifically, this way USE can help a policy designer to detect certain conflicts between authorisation constraints and find missing constraints. Secondly, USE can not only be employed in the design phase of a security policy, but also after constraint deployment. In this respect, USE may be helpful for external official reviewers. Finally, the Java API provided by USE can be facilitated to build an authorisation editor which can enforce various types of authorisation constraints.

Owing to the fact that USE can only check the current snapshot of an RBAC configuration, history-based authorisation constraints [25] cannot be dealt with. For this purpose a temporal extension of OCL like that sketched in this paper is needed. Hence, it remains future work to extend USE in order to deal with temporal constraints. This way, the authorisation editor can enforce history-based constraints, too. Another goal is to integrate the authorisation editor into middleware. Specifically, Web services could be an interesting target to enforce authorisation constraints due to the high access control requirements of this technology.

## 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. R. Anderson, *A security policy model for clinical information systems*, Proceedings of the IEEE Symposium on Research in Security and Privacy (Oakland, CA), IEEE Computer Society Press, May 1996, pp. 30–43.
5. D. D. Clark and D. R. Wilson, *A comparison of commercial and military computer security policies*, Proceedings of the 1987 IEEE Symposium on Security and Privacy (1987), 184–194.

6. J. Crampton, *Specifying and enforcing constraints in role-based access control*, Proc. of the 8th ACM Symposium on Access Control Models and Technologies (New York), ACM Press, June 2–3 2003, pp. 43–50.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Handbook of graph grammars and computing by graph transformation, Vol. 2: Applications, Languages and Tools*, World Scientific, Singapore, 1999.
8. EU, *Directive on the protection of individuals with regard to the processing of personal data and on the free movement of such data. Directive 95/46/EC. http://www.privacy.org/pi/intl_orgs/ec/eudp.html*, 1995.
9. D. Ferraiolo, D. Gilbert, and N. Lynch, *An examination of federal and commercial access control policy needs*, Proc. of the NIST-NCSC Nat. (U.S.) Comp. Security Conference, 1993, pp. 107–116.
10. D.F. Ferraiolo, D.R. Kuhn, and R. Chandramouli, *Role-based access control*, Artec House, Boston, 2003.
11. C.K. Georgiadis, I. Mavridis, G. Pangalos, and R.K. Thomas, *Flexible team-based access control using contexts*, Proc. of the ACM Symposium on Access Control Models and Technologies, May 3–4 2001, pp. 21–27.
12. V. D. Gligor, S. I. Gavrila, and D. Ferraiolo, *On the formal definition of separation-of-duty policies and their composition*, 1998 IEEE Symposium on Security and Privacy (SSP '98), IEEE, May 1998, pp. 172–185.
13. M. Gogolla, J. Bohling, and M. Richters, *Validation of UML and OCL Models by Automatic Snapshot Generation*, Proc. 6th Int. Conf. Unified Modeling Language (UML'2003), Springer, Berlin, LNCS 2863, 2003, pp. 265–279.
14. Martin Gogolla and Mark Richters, *Transformation Rules for UML Class Diagrams*, Proc. 1st Int. Workshop Unified Modeling Language (UML'98), Springer, Berlin, LNCS 1618, 1999, pp. 92–106.
15. T. Jaeger and J.E. Tidswell, *Practical safety in flexible access control models*, ACM TISSEC **4** (2001), no. 2, 158–190.
16. M. Koch, L. V. Mancini, and F. Parisi-Presicce, *A Graph Based Formalism for RBAC*, ACM Transactions on Information and System Security (TISSEC) **5** (2002), no. 3, 332–365.
17. M. Koch and F. Parisi-Presicce, *Visual Specification of Policies and their Verification*, Proc. of Fundamental Approaches to Software Engineering (FASE) 2003, LNCS, no. 2621, Springer, 2003, pp. 278–293.
18. T. Mossakowski, M. Drouineaud, and K. Sohr, *A temporal-logic extension of role-based access control covering dynamic separation of duties*, Proc. of TIME-ICTL 2003, Cairns, Queensland, Australia, July 8–10 2003.
19. M. J. Nash and K. R. Poland, *Some conundrums concerning separation of duty*, Proc. IEEE Symposium on Research in Security and Privacy, 1990, pp. 201–207.
20. 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.
21. 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.
22. M. Richters and M. Gogolla, *Validating UML Models and OCL Constraints*, Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000), Springer, Berlin, LNCS 1939, 2000, pp. 265–277.
23. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual, Second Edition*, Object Technology Series, Addison Wesley Longman, Reading, Mass., 2004.

24. 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.
25. 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.
26. 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.
27. J. Warmer and A. Kleppe, *The Object Constraint Language: Getting your models ready for MDA*, Addison-Wesley, Reading/MA, 2003.
28. L. Zhang, G.-J. Ahn, and B.-T. Chu, *A role-based delegation framework for healthcare information systems*, Proc. of the 7th ACM Symposium on Access Control Models and Technologies (New York), June 3–4 2002, pp. 125–134.
29. P. Ziemann and M. Gogolla, *An OCL Extension for Formulating Temporal Constraints*, Research Report 1/03, Universität Bremen, 2003.