Enforcing Role-Based Access Control Policies in Web Services with UML and OCL

Abstract Role-based access control (RBAC) is a powerful means for laying out and developing higher-level organizational policies such as separation of duty, and for simplifying the security management process. One of the important aspects of RBAC is authorization constraints that express such organizational policies. While RBAC has generated a great interest in the security community, organizations still seek a flexible and effective approach to impose role-based authorization constraints in their security-critical applications. In this paper, we present a Web Services-based authorization framework that can be employed to enforce organization-wide authorization constraints. We describe a generic authorization engine, which supports organization-wide authorization constraints and acts as a central policy decision point within the authorization framework. This authorization engine is implemented by means of the USE system, a validation tool for UML models and OCL constraints. Using this system, we also demonstrate how the authorization constraints can be specified in OCL and then be implemented by USE.

1 Introduction

Employing access control mechanisms in medium to large scale organizations always has been crucial. One of the challenging jobs for security-critical organizations, such as financial institutes, hospitals and, military is to control access to system resources at the highest level without violating the underlying access control policies. The research in recent years has brought role-based access control (RBAC) [3, 4] as an efficient and flexible model for controlling access to computer resources and enforcing the organizational policies. According to our terminology, an organizational policy consists of a set of organizational rules. A typical organizational rule in a hospital might be “a nurse can only see the records of all patients who have been on her ward within the previous 90 days”. Similarly, in banking applications, a rule might be “a clerk must not prepare and approve a check”.

As pointed out by Ferraiolo et al. [5, 21], one of the main advantages of RBAC is that such higher-level organizational rules can be implemented in a natural way. Specifically, advanced RBAC concepts like role-based authorization constraints are a powerful means for laying out higher-level organizational rules [7]. Hence, we define an RBAC policy as hierarchical RBAC in the sense of the RBAC standard [14] plus a set of organizational rules where each rule corresponds to a role-based authorization constraint, such as separation of duty (SOD) constraints [7, 8, 9], cardinality constraints [3], and context constraints [10, 11].

Given the fact that an organization will be running a number of different applications (assuming legacy applications as well), employing an approach to enforce different organization-wide RBAC policies is still an open problem. Designing and implementing such an RBAC system raises many critical questions, of which some of them are: (1) how can we separate authorization logic (access control mechanisms) from the application logic (application code) to make organization-wide RBAC policies easier to administer?, (2) how can a platform- and application independent authorization engine be developed that implements organization-wide RBAC policies?, (3) how much of the authorization logic can be abstracted out of the application code?, and last but certainly not the least, (4) what technological approach can we adapt that facilitates a flexible integration of organization-wide authorization components and various applications?

In this paper, we present an authorization framework as a first step towards the solution to the aforementioned problems. This framework encompasses specification, implementation and, specifically, the enforcement of organization-wide RBAC policies.

Firstly, we show an implementation of a platform- and application independent authorization engine that implements organization-wide RBAC policies. This way the RBAC policies can be centrally administered. The authorization engine is based on the USE system [16], a validation tool for UML models and OCL (Object Constraint Language) constraints. In general, the UML [1] and OCL [2] can be used in combination for designing and developing software systems. We use UML/OCL specifically for the specification of RBAC policies in our authorization framework. The OCL approach is formal and precise, and can express various kinds of authorization constraints, such as static and History-based SOD constraints, cardinality constraints and context constraints. The authorization engine, in general, is powerful enough to specify and implement all authorization constraints that are expressible in OCL. Moreover, the USE tool itself can be employed to validate RBAC policies formulated in UML and OCL.

Secondly, we present an advanced Web Services-based RBAC authorization framework, which can be employed to enforce organization-wide RBAC policies across various applications. Owing to the fact that Web Services aim at integrating various applications of an organization the enforcement of organization-wide RBAC policies at the Web Service (middleware) level is an important task to simplify access management. In particular, organizations integrate more and more applications (including legacy applications) by means of Web Services and make available Web-Service interfaces to expose specific (often security-critical) functionality of such applications. For example, the functionality of a credit rating application might be exposed to clerks in branches of a financial institute [28]. Due to the fact that specifically legacy applications often do not have adequate access control mechanisms, our proposed authorization approach helps in improving security within organizations by enforcing organizational rules.

Our authorization framework is based on the concept of an *interceptor*, a middleware component. The interceptor is used to integrate the organization-wide authorization engine and various application(s) into the middleware by means of Web Services. In general, the interceptor plays a central role to enforce organization-wide RBAC policies. One of the benefits of this approach is that the application does not need to contain any authorization logic and any change of the RBAC policy does not require any modifications of the application.

We implemented a prototypical interceptor for Java applications that are integrated over the Internet by SOAP-based Web Services. In addition, we demonstrate how the authorization framework copes with the specification, implementation and enforcement of authorization constraints through various examples. This way, our approach combines well-understood concepts of a widely used modeling language with the Web Service technology to implement advanced RBAC mechanisms for organizations.

The rest of the paper is organized as follows: in Section 2 we provide a brief overview of related concepts and technologies. Section 3 provides an overview of the authorization framework and a detailed description of the framework components follows. In Section 4, we illustrate the overall functionality of the implemented framework through various examples. In Section 5, we use case studies to demonstrate that our authorization engine can deal with authorization constraints from various domains, and how such constraints are enforced within our authorization framework. An overview of related work is given in Section 6. We outline our conclusions and future work in Section 7.

2 Related Concepts and Technologies

2.1 RBAC and Authorization Constraints

RBAC [3, 4] has gained much attention as a promising alternative to traditional discretionary and mandatory access control. It is an access control model in which the security administration can be simplified by the use of roles to organize the access privileges and ultimately reduces the complexity and cost of security administration [5]. Here we give an overview of the components of RBAC96, a widely used RBAC model introduced by Sandhu et al. [3]:

- the sets U, R, P, S (users, roles, permissions, and sessions, respectively)
- $UA \subseteq U \times R$ (user to role assignment relation)
- $PA \subseteq P \times R$ (permission to role assignment relation)
- $RH \subseteq R \times R$ is a partial order called the role hierarchy relation.

A user can be a member of many roles and a role can have many users. Similarly, a role can have many permissions and the same permissions can be assigned to many roles. A user may activate a subset of roles he or she is assigned to in a *session*. The permissions available to the users are the union of permissions from all roles activated in that session. Role hierarchies can be formed by the *RH* relation. Senior roles inherit permissions from junior roles through the *RH* relation (e.g., the role *chief physician* inherits all permissions from the *physician* role).

Authorization constraints are an important aspect of RBAC and are sometimes considered to be the principle motivation behind RBAC. The goal of authorization constraints is not only to reduce the risk of fraud or a security breach but to increase the opportunity of detecting errors within an organizational security structure. Authorization constraints may need to be imposed on the RBAC functions and relations in order to prevent the information misuse and fraudulent activities. In the literature, several kinds of authorization constraints have been identified such as various types of static and dynamic SOD constraints [7, 8, 9]; constraints on delegation [10]; cardinality constraints [3]; context constraints [10, 11].

Specifically, SOD is a fundamental principle in security systems and is typically considered as a requirement that, operations are divided among two or more persons so that no single individual can compromise the security. SOD constraints are used to enforce conflict of interest policies. One means of preventing conflict of interest is through static SOD, that is, to enforce constraints on the assignment of users to roles. On the other hand, the dynamic SOD constraints limit the permissions that are available to a user by placing constraints on the roles that can be activated within or across a user's sessions.

2.2 UML and OCL

UML is a general-purpose modeling language in which we can specify, visualize, and document the components of software systems [1]. It captures decisions and understanding about software 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 of software systems. In this paper, we concentrate on the static UML models.

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 static UML model for RBAC consisting of the RBAC classes and associations is depicted (UML class diagram). The classes and associations correspond to the RBAC sets and relations defined in Section 2.1. A further UML diagram type relevant to our work is the object diagram. Here, objects are instances of classes and links are instances of associations. An object diagram then provides a snapshot of a system at a particular point in time showing objects, their attribute values, and links connecting the objects [1].

OCL [2] 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. 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. Invariants are conditions that must be true during the lifetime of a system for all instances of a given type. The following line shows an example of an OCL invariant describing a role with at most one user:

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

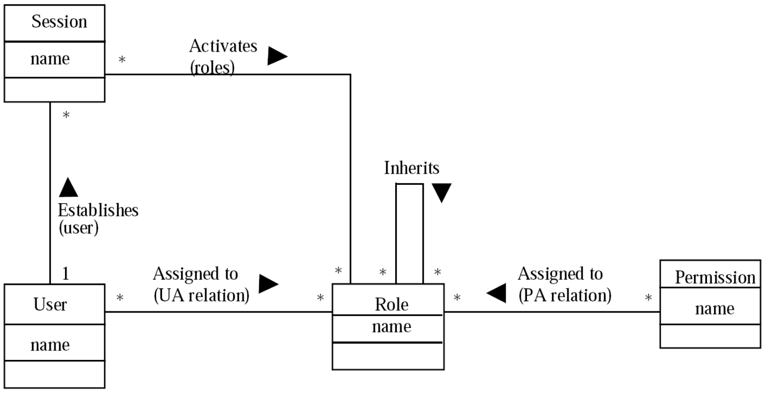


Fig. 1. Class Model for RBAC-Entity Classes.

`self` refers to an instance of `Role`. Then `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`[1]:

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

The expression `self.role_->includes(r1)` means that `r1` 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 iterate over the members of a collection (set, bag, sequence) such as `forAll`, `exists`, `iterate`, and `any`. These operations are used several times throughout this paper.

3 Authorization Framework

We have designed and implemented an advanced Web Services-based authorization framework for the enforcement of organization-wide RBAC policies. In our authorization framework, the authorization engine and the application(s) are exported as Web Services. The authorization engine implements organization-wide RBAC policies and makes access decisions.

The authorization framework is based on the concept of an *interceptor* (Access Decision Handler), a middleware component. Specifically, the interceptor en-

[1] The word "role" is a keyword in the USE format, which is introduced later in the paper. In order to distinguish the USE keyword from the word "role" in the sense of RBAC, we append an underscore for this meaning of the word "role".

forces the access decisions on behalf of the applications. This way, the applications do not need to implement their own authorization mechanisms. Instead, they can employ the authorization engine to conduct the job accordingly. As a consequence, RBAC policies such as SOD need not be implemented in the application itself, i.e., applications can rely on our framework for this purpose. Furthermore, if RBAC policies are changed, the applications need not be adjusted. This leads to a separation of the authorization logic (integrated into the middleware) from the application logic. Since the organizations may be running legacy applications, the Web Services approach is suitable to implement interceptors even for legacy applications, while still using the centralized authorization engine.

In Figure 2, an overview of the authorization framework is given. The communication between the framework components is based on SOAP messages. The *Authorization Engine* is a policy decision point (PDP), whereas the *Access Decision Handler* is a policy enforcement point (PEP) [15]. The PDP evaluates client requests against relevant RBAC policies to return an authorization decision. The PEP enforces the PDP's decisions. Details are discussed subsequently.

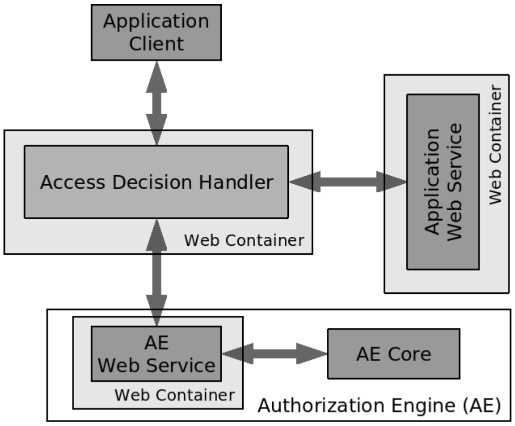


Fig. 2. Authorization Framework.

3.1 Details of the Authorization Framework

Figure 3 shows a sequential view of the authorization and communication between the interacting components. When an *Application Client* calls a security-critical operation such as "debit account" on the *Application Web Service*, the *Access Decision Handler* intercepts the call request and forwards it to the *Authorization Engine* in order to check whether or not the client has the permission to perform the current operation. The *Authorization Engine*, on receiving the request from the *Access Decision Handler*, decides if the client has a permission to carry out the current operation, and sends the response back to the *Access Decision Handler*. In turn, the *Access Decision Handler* enforces the access decision of the *Authoriza-*

tion Engine by allowing or rejecting the client's request to perform the current operation on the *Application Web Service*. This way, the interceptor can be seen as a mediator [17] between the Web-Service-based applications and the *Authorization Engine*.

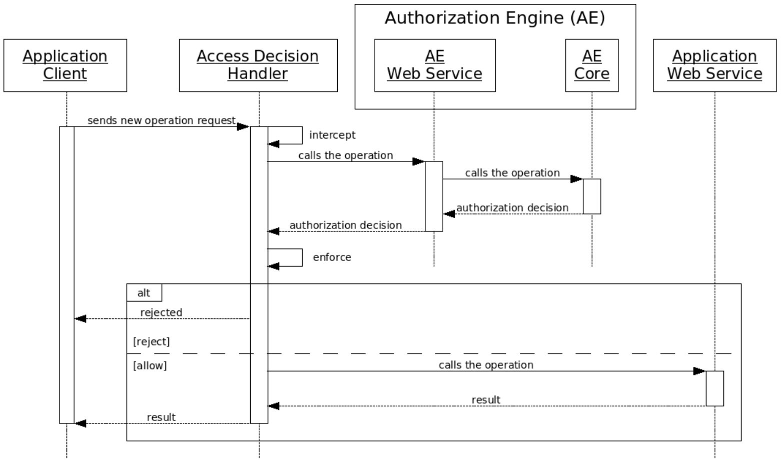


Fig. 3. Authorization Flow and Communication Path.

The *Access Decision Handler* (interceptor) has been realized the following way: For the *Application Client*, it acts as a Web Service (exposing the interfaces of the *Application Web Service* and of the *Authorization Engine* to the *Application Client*). At the same time, the interceptor acts as a Web Service client for both the *Application Web Service* and the *Authorization Engine*.

A further aspect to be discussed is session management. We use the session management provided by the Web container (Apache's Axis in our case) for this purpose: When a user logs in via the *Application Client*, an http session ID is created by the Web container. This session ID is then communicated to the PDP, which uses this information for creating its own internal representation of RBAC sessions (cf. Section 4).

One remark should be made on the mapping between the *Application Web Service* interface and the RBAC permissions, which are used by the *Authorization Engine* for the access decision. We define the remote interfaces for the *Application Web Service* in a way that all methods have exactly one parameter, namely, the object (in the sense of access control) to which the operation/method is applied. For example, if a debit operation is to be executed on an account object, we have a Java method *debit* with the parameter *account*. This way, we can create corresponding RBAC permissions such as *(debit, account)* and thus have a mapping from the *Application Web Service* interface to RBAC permissions. The interceptor then extracts the operation (method call) and object (parameter). We assume that the *Application Web Service* interface is defined by domain experts knowing the organization's internal rules and processes in detail. The same applies to the definition of the RBAC permissions.

3.2 Trust Considerations

Strictly speaking, the communication channel from the *Application Client* to the *Application Web Service* must be secured. This can be done by means of XML Digital Signature [27]. In particular, we need a secure channel between the PDP (*Authorization Engine*) and the PEP(s). At least, the *Access Decision Handler* (PEP) and the PDP must trust each other. Furthermore, the *Application Web Services* must trust the PEP and by transitivity the PDP. The *Application Client*, however, is not trustworthy because it is under control of the user (and possibly under control of an attacker). If the PEP and PDP need feedback from the *Application Web Service*, then a trust relationship in that direction must also exist. This, for instance, is the case if the access history (needed for implementing History-based SOD [9]) must be updated after an operation has been successfully carried out by the *Application Web Service*. Due to the fact that we concentrate here on the implementation of RBAC policies and the authorization engine, we do not deal with trust aspects in this paper in more detail.

4 Functionality of the Authorization Engine

The authorization engine is the core component of the authorization framework, which implements organization-wide RBAC policies. Such policies are eventually enforced by means of the authorization framework. We have implemented an advanced authorization engine, which is based on the Java API provided by the USE system, a validation tool for UML models and OCL constraints [16].

We use the UML and OCL specifications provided by USE to formulate RBAC policies. Specifically, the RBAC element sets and relations are modeled in textual UML, and the authorization constraints are specified in OCL. Owing to the fact that OCL can be used to express the authorization constraints formally and precisely, a validation tool such as USE can be applied to recognize violations of such constraints. Hence, one advantage of our approach is that USE can be employed both for validation and enforcement of RBAC policies.

The USE system is based upon a so-called animation-based validation approach, i.e., the OCL constraints are checked against **system states**, which are represented as UML object diagrams [1]. Beyond syntax checks, USE supports the modeler in detecting missing or conflicting constraints. For the sake of brevity, the validation of RBAC policies, however, is not topic of this paper. The interested reader may be referred to another paper [22].

In Figure 4, we show a simple RBAC policy, which is represented as a USE specification. It consists of the RBAC-related classes and association definitions formulated in textual UML, and a set of authorization constraints formulated in OCL. We define three constraints. The first is a prerequisite role constraint be-

```
model RBAC

--classes

class Role
attributes
name:String
end

class User
attributes
name:String
end

class Permission
attributes
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
User[1] role user
Session[*] role session
end

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

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

Constraints

-- Prerequisite roles
context User inv PrerequisiteRole:
self.role_->includes(Cashier)
implies self.role_-> includes(Banking_Employee)

-- Static SOD - Conflicting Users
context Role inv SSOD-CU:
let
  CU:Set(User)=Set{Frank,Joe},
  CR:Set(Role)=Set{Cashier, Cashier_Supervisor}
in
CU->iterate(u:User;
    result:Set(Role)=oclEmpty(Set(Role))|
    result->union(u.role_))->
               intersection(CR)->size()< CR->size()

-- Simple Dynamic SOD
context User inv SDSOD:
let
  CR:Set(Role)=Set{Customer,Cashier}
in
CR->intersection(self.session->iterate(s:Session;
      result:Set(Role)=oclEmpty(Set(Role))|
      result->union(s.role_)))->size()< CR->size()
```

Fig. 4. USE specification of an RBAC policy.

tween two roles “Banking Employee” and “Cashier”, i.e., a user must be assigned to the “Banking Employee” role if she is assigned to the “Cashier” role. The second one is an SSOD-CU constraint (Static SOD-Conflict Users) meaning that conflicting users cannot be assigned to conflicting roles. A typical example of conflicting users is family members who could collude to commit fraud. The third constraint is the Simple Dynamic SOD (SDSOD) constraint [9], which states that a user must not activate the “Customer” and the “Cashier” role simultaneously. Technically, CU and CR denote sets of conflicting users and roles, respectively.

The RBAC policy shown in Figure 4 is only for didactic purposes, which by no means is a complete policy that the authorization engine implements. For example, we left out the OCL constraints representing the partial order conditions of role hierarchies.

In general, the authorization engine is independent of any specific UML/OCL-based RBAC model, and, as pointed out before, the authorization engine implements all authorization constraints expressible in OCL. The RBAC policy is saved in a file in the USE format, which is processed when the authorization engine is started. This way, a security officer specifies the RBAC policy in UML/OCL and the USE system takes over the job of implementing the policy.

The specification in Figure 4 has the drawback that the concrete entities (such as users and roles) are hard-coded into the specification. In addition, the policy designer might not be an expert in OCL. However, a macro/template mechanism can be provided in order to make available recurring types of authorization constraints for a policy designer. These macros are then instantiated with the concrete entities on which the authorization constraints are to be applied. This resembles the macro mechanism introduced into the C programming language. Let us take the prerequisite role constraint as an example. Then, we can define the macro `PrereqRole(_r1_,_r2_)` as follows:

```
context User inv PrerequisiteRole:
      self.role_->includes(_r2_)
                  implies self.role_->includes(_r1_).
```

We can then instantiate this macro with the actual parameters of the authorization constraint in question. For instance, the macro call `PrereqRole(Banking_Employee, Cashier)` will then be expanded by a macro pre-processor to the prerequisite role constraint displayed in Figure 4.

4.1 RBAC Administrative and System Functionality

The authorization engine supports most of the functionality demanded by the ANSI RBAC standard [14], i.e., it also contains functionality of a policy administration point (PAP) [15]. In particular, we have implemented administrative functions, review functions, and system functions. Administrative functions (e.g., *AddUser*, *AssignUser*) are required for the creation and maintenance of the RBAC element sets and relations. Review functions (e.g., *AssignedRoles*, *UserPermissions*) can be employed to inspect the results of the actions created by administrative functions. System functions such as *CreateSession*, *AddActiveRole*, and *CheckAccess* are required by the authorization engine for session management and making access control decisions.

The administrative RBAC functions and some of the system functions such as *CreateSession* are implemented by the USE component which is responsible for creating system states (animator). This way, system/security states are built, which are employed by USE (authorization engine) for making access control decisions.

In addition, we use OCL expressions to specify the RBAC review and the remaining RBAC system functions. As a consequence, not only are the authorization constraints specified in OCL, but also parts of the administrative functionality. For example, the review function *UserPermissions*, which returns all the permissions belonging to a user, is specified as follows in OCL:

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

This OCL specification is then simply called by the `eval` method made available by the Java API of the USE system. This allows a developer to specify main parts of the authorization engine in UML/OCL and then obtain an implementation semi-automatically, with only a few adjustments (e.g., concerning the GUI) [22].

4.2 Advanced RBAC Concepts

One of the important aspects of the authorization engine is to incorporate advanced RBAC concepts that are comprised of various kinds of authorization constraints. We now demonstrate that the authorization engine can deal with a diverse range of authorization constraints. In particular, far more constraints are supported than the simple SOD constraints of the RBAC standard [4], which is deliberately restricted to common RBAC concepts.

The authorization constraints can be specified in OCL in a similar way as shown in Figure 4. However, due to space restrictions, the OCL specifications for most of the constraints are left out in the following discussion.

4.2.1 Authorization Constraints

In this section, we give only few examples of authorization constraints, which can be realized with the help of the authorization engine. More specific examples of authorization constraints from different domains are given in Section 5.

The authorization engine supports various SOD constraints. One example has already been mentioned, namely, SSOD-CU (cf. Figure 4). Other constraints are simple static SOD [4] or conflicting permissions [7]. The conflicting permissions constraint, for example, states that the same user may under no circumstances receive the "approve order" and "approve audit" permission. We also realized the SDSOD constraint given in Figure 4, and a session-based dynamic SOD by which no user can activate two conflicting roles *within a single session*.

The authorization engine also supports cardinality constraints and prerequisite constraints. The cardinality constraints are instructions of the form "a department has exactly one chairperson", or "every document must have at least one author". More generally, Sohr et al. [22] show that the authorization engine can handle all the types of authorization constraints that can be formulated in RCL 2000, a well-known specification language for RBAC authorization constraints [7].

One remark should be made on the implementation of authorization constraints by means of the USE system. This is done by calling the `check` method provided by the Java API of the USE system. Similarly to the implementation of the review functions (cf. Section 4.1), the constraint is passed as a parameter to the `check` method. If the authorization constraint to be checked is satisfied in the current system state, the `check` method returns true, otherwise false. Hence, we obtain a simple implementation of authorization constraints and use the functionality of a general-purpose UML-/OCL validation tool for the evaluation of RBAC policies. In particular, this applies to all authorization constraints discussed in the course of this paper, and in particular, to the constraints of the case studies (cf. Section 5).

4.2.2 History-Based SOD

The authorization engine also supports different forms of History-Based SOD. One example is Object-Based Dynamic SOD (ObjDSOD) as proposed by Simon and Zurko [9]. ObjDSOD states that a user may act upon an object with at most one critical operation. For example, a user may only perform one of the operations prepare, approve, and sign on a check.

In order to formulate such history-based constraints, we introduce a further attribute called `accesshistory` for each object. This attribute represents the access history of the object in question. Subsequently, we give an example specification of the ObjDSOD constraint for the check object:

```
context User inv ObjDSoD:
let
check:Object=Object.allInstances->any(name ='check'),
crit_ops:Set(Operation)=Set{prepare,approve,sign}
in
 check.accesshistory->forAll(t1,t2|
    (t1.u=self and t2.u=self
    and crit_ops->includes(t1.op)
    and crit_ops->includes(t2.op))
                       implies t1.op=t2.op)
```

The access history is a sequence of pairs (user, operation), i.e., each access on the object in question is logged in the history. Moreover, note that we use OCL's tuple types to represent the access pairs. Through the use of OCL sequences ordered-dependent history-based SOD constraints [9] can be formulated, too. The aforementioned OCL constraint states that if a user has executed two (critical) operations on a check, both operations must be the same.

History-based constraints ought to be enforced at runtime when an operation is executed on an object. Then we would have to change the history information *after* the operation has been successfully executed. However, this would mean that we need a feedback from the *Application Web Service*. Specifically, if the *Application Web Service* is not reachable, that security-relevant information will not be communicated back to the PDP. For this reason, we decided to enforce those constraints whenever an access request is to be checked, i.e., on calling *CheckAccess*. Furthermore, we decided to store the access control history for each object in the security state (USE system state) and not at the object itself. This means that we must also update the access control history in the USE system whenever *CheckAccess* is successfully called.

4.2.3 Context Constraints

Context constraints [10, 11, 13] are another variation of authorization constraints, which allow organizations to restrict access to data or business processes according to the contexts such as location and time. The authorization engine supports context-based permission activation [23]. Context information (like location and time) allows one to express a variety of authorization constraints that can further tighten the permission activation. 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. A context constraint w.r.t. location can now be formulated in OCL as follows:

```
context User inv LocationContext:
    self.session->forAll(s|s.role_->forAll(r|
          r.permission->forAll(p|
                ActivePermission(self,r,p) and
                p.o.location=self.location))).
```

First, we introduced a ternary association `ActivePermission`[2] stating that the user (`self`) has activated permission `p` with role `r`. Second, we assume that both the user and the object have an additional attribute "location" that describes the current location of the user and the object, respectively. If now a role is activated in a session, then the location of the permission (or more exactly, of the permission's object) must be the same as the location of the user. The aforementioned context constraint is then checked by the PDP when the RBAC system functions *AddActiveRole* or *CreateSession* are called.

These kinds of context constraints often occur in hospitals as will be mentioned in Section 5.3. However, we do not claim here that we have implemented an elaborated mechanism for obtaining the context information from a source such as a directory service or a personnel system. This again lies outside the scope of this paper that concentrates on the implementation of an authorization engine and its integration into a Web Service environment.

5 Case Studies

As mentioned above, the authorization engine implements various RBAC policies, i.e., the authorization engine is independent of the domain. This will be demonstrated by three case studies from different domains, namely, banking, military, and health care. Note that in all these cases the RBAC policy in question is defined in a USE file. This is similar to the RBAC policy depicted in Figure 4, containing the domain-specific authorization constraints. The USE system then processes this file and enforces the RBAC policy as sketched in Section 4.2.1.

5.1 Separation of Duty in a Banking Environment

The first case study reflects the fact that Web Services are used in financial institutes to integrate applications [29]. In particular, SOD often occurs in those applications. We specifically implemented the SOD constraints shown in Figure 4 such

[2] Ternary associations can be expressed in OCL with association classes. However, as we have done earlier [23], we again introduce an additional ternary predicate to simplify the discussion.

as the SSOD-CU constraint which prevents two colluding users to be assigned to the conflicting roles “Cashier” and “Cashier Supervisor”. Due to the fact that SSOD-CU is static, the constraint is directly enforced by the authorization engine at administration time. Whenever the administrative RBAC functions such as *AssignUser* are called, the authorization engine checks the already defined static authorization constraints. The other parts of our authorization framework (cf. Figure 2) are not involved in enforcing such static authorization constraints.

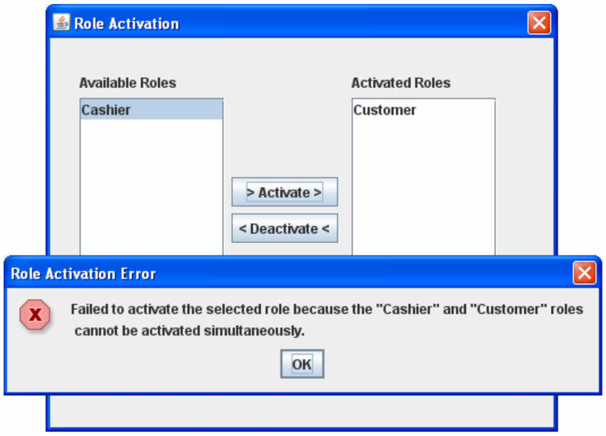
Dynamic SOD constraints have been implemented, too, such as the SDSOD constraint between the roles “Cashier” and the “Customer”. Figure 5 shows a dialog window made available by the Banking application client prototype for activating roles. Here, a user has already activated the “Customer” role. Activating the “Cashier” role is forbidden due to the SDSOD constraint as shown in Figure 5.

Considering our framework introduced in Section 3, the following steps are carried out. The user tries to activate the “Cashier” role in the *Application Client*, i.e., the RBAC system function *AddActiveRole* is called (cf. Figure 2). This request is passed from the *Application Client* via the *Access Decision Handler* to the authorization engine. The engine then makes the access decision based upon the current security/system state (in our example, the “Customer” role has already been activated) and the already defined authorization constraints (SDSOD in our case). The engine then returns the decision “activation forbidden” back to the *Access Decision Handler*. In turn, the *Access Decision Handler* communicates the result to the banking application client as shown in Figure 5. Had the activation been permitted, the security state within the authorization engine would have been changed accordingly.

Note that in this particular case the banking application Web Service itself never sees the activation process. The role activation is actually done inside the authorization engine. Having activated appropriate roles, certainly the user of the banking application client can carry out different operations on the banking application Web Service such as debit account, and consequently other dynamic authorization constraints can come into effect, such as context constraints.

Within the frameworks of this case study, we carried out early performance measurements. In particular, we measured the time for executing access requests from the *Application Client* to the *Application Web Service*. For this purpose, we configured an RBAC system with 50 users and 10 roles. The interceptor runs on a Pentium IV, 2 GHz, and the *Authorization Engine* on a Pentium M, 1.6 GHz. Executing 1000 random access requests such as *debit/credit account*, we obtained an average latency time of about 86 ms per access (compared to 14 ms per access without using our framework). This way, an access request is about six times slower than an ordinary Web Service request without employing our authorization framework.

Our authorization framework, however, can be configured in a way that only security-critical access requests are guided through our framework (cf. Section 3.1). Other requests need not be intercepted, i.e., functionality which is not security-critical is exported directly by the *Application Web Service* and not by the interceptor. Remember that we assume that the *Application Web Service* interface is designed by a domain expert who knows which functionality is security-critical and which not.



Failed to activate the selected role because the "Cashier" and "Customer" roles cannot be activated simultaneously.

Fig. 5. Enforcing SDSOD in a banking application client.

5.2 Implementing Lattice-Based Access Control Policies

Our authorization framework can also be employed for implementing lattice-based access control (LBAC) policies [24]. LBAC has specifically been used in military, but sometimes LBAC has also been implemented in large enterprises [25]. Although LBAC is an old access control model, we use it here to demonstrate the flexibility of the authorization engine component of our authorization framework.

As shown by Sandhu [26], RBAC can be configured to simulate LBAC policies by forming two dual role-hierarchies (one for the read roles, and the other one for the write roles) and by defining authorization constraints on the RBAC relations. In particular, a security label x is then represented by two roles *xR* and *xW* with *xR* the appropriate read role and *xW* the appropriate write role. For the sake of brevity, we leave out here the details of this construction. The interested reader is referred to the work by Sandhu [26].

For demonstration purposes, we now discuss a simple LBAC policy with two security labels low and high. In order to employ the USE system for implementing that LBAC policy, we first express the policy in OCL. To give an impression, how the OCL version of the policy looks like, we give here one of the authorization constraints used by Sandhu; the other constraints of Sandhu's construction can be expressed similarly. The constraint states that each session has exactly two roles *xR* and *xW* and means that the user has logged in at the security level x. For this purpose, we introduce two role sets RR = {L1R, …, LnR} and WR = {L1W, …, LnW} for the read and the write roles, respectively. Furthermore, we add an attribute called `label` for each role. Then, we obtain the following OCL constraint:

```
context Session inv SessionConstraint:
let
  RR : Set(Role) = Set{L1R, …, LnR},
  WR : Set(Role) = Set{L1W, …, LnW}
in
  self.role_->size()=2 and
  self.role_->forAll(rr, wr|RR->includes(rr) and
    WR->includes(wr) and rr.label=wr.label)
```

Note that this specific authorization constraint is enforced at run-time and not at administration time because sessions are involved. The constraint is checked whenever a new user session is created by the *Application Client* (cf. Section 3), which tries to access classified data. In this case, the *CreateSession* RBAC system function is called by the *Application Client*. This request is then passed to the authorization engine via the *Access Decision Handler*. If the constraint is violated, this result is returned to the *Application Client*. Otherwise, the security state is changed within the authorization engine accordingly, i.e., a new object of the UML class `Session` is generated. In addition, the fact that the role has been activated successfully must also be stored in the authorization engine.

Having specified the entire LBAC policy in a USE file (also including the other four authorization constraints given by Sandhu's LBAC construction), we have implemented LBAC by means of the USE system.

5.3 Context Constraints in a Hospital Information System

Owing to the fact that hospital information systems (HIS) often consist of different modules (e.g., for radiology and cardiology) Web Services can help to integrate those modules. Due to the strict privacy rules, however, advanced access control mechanisms for Web Services are important for the health care domain.

We have prototypically realized the authorization framework, in which the authorization engine and a prototype for a HIS communicate with each other by means of an interceptor (cf. Figure 2). We also have implemented a HIS client to demonstrate how the authorization framework enforces context constraints w.r.t. location (cf. Section 4.2.3). On logging in to the HIS, a clinician only obtains the permissions of the ward she is currently assigned to. For example, the clinician can only read electronic health records (EHRs) of patients who are currently being treated in the clinician's ward. Since this context constraint is applied during the log-in process, the constraint is checked when *AddActiveRole/CreateSession* are called. This is done similarly as described in the sections 5.1 and 5.2.

Whenever the clinician now attempts to read an EHR through the HIS client, the request is intercepted by the *Access Decision Handler* component of the authorization framework and passed on to the authorization engine acting as a PDP. The PDP then decides whether the clinician can execute the read operation on the patient's EHR. Note that the clinician's ward and the patient's ward need not be checked again because this constraint has already been checked at role activation time. The PDP's authorization decision is finally enforced by the *Access Decision Handler* (PEP). If the clinician has the appropriate permissions, the request is forwarded to the HIS Web Service (*Application Web Service*). Otherwise, the request is denied and this is also reflected on the HIS client side, i.e., in the *Application Client* (cf. Figure 3).

6 Related Work

There is a plethora of works in the context of security modeling with UML such as [18, 19, 29]. As indicated above, the USE system is a general-purpose validation tool and can hence be employed for other UML/OCL encodings of RBAC policies than that given in Section 4. In particular, Lodderstedt et al. present the modeling language SecureUML for integrating the specification of access control into application models and automatic generation of access control infrastructures for applications [18]. They also deal with authorization constraints, but do not concentrate on SoD constraints. Another difference is that our aim was to make available an authorization engine for Web Service applications that can enforce various RBAC policies. Then, applications can use our authorization framework if needed. We not primarily intended to provide a methodology to integrate access control and specific application models.

XACML is an OASIS standard that supports the specification of authorization policies and related queries in a standardized, machine-readable way [15]. The RBAC profile of XACML 2.0 extends the standard for expressing authorization policies that use RBAC with a scope limited to core and hierarchical RBAC [20]. UML/OCL, however, is a standard modelling approach that can be used to express the RBAC policies in a human-readable way. Specifically, OCL can be used to express various kinds of role-based authorization constraints, whereas the RBAC profile of XACML 2.0 lacks the full support of SOD constraints and other variations of authorization constraints. It could be argued that RBAC policies can be specified directly in XACML. However, manually specifying such policies directly in XACML could be comparatively complicated, time consuming and hence error-prone. Due to the fact that we specify RBAC policies in a formal modelling language (UML/OCL) we can validate the policies w.r.t. conflicting and missing constraints by tools such as USE [22]. To our knowledge, there currently do not exist any tools that are able to validate XACML policies in that sense.

Furthermore, there exist a plethora of authorization engine prototypes, which can be compared with our engine. One of those engines is Adage, developed by Zurko et al. [30]. Adage has been developed with similar goals in mind. Specifically, Adage can enforce different kinds of role-based SOD constraints on the middleware layer (for example, Adage was integrated into CORBA). Moreover, Adage makes available a policy specification language called AL. However, some constraint types are not supported by Adage such as context constraints. Furthermore, a validation tool for access control policies is not available. Bhatti et al. present an authorization framework for Web Services which can enforce temporal constraints in the sense of the GTRBAC model introduced by Joshi et al. [31]. In addition, simple SOD constraints are supported. Crampton presents another authorization engine [32]. However, with Crampton's approach, for example, neither the SSOD-CU constraint nor context constraints can be enforced.

7 Conclusion and Future Work

In this paper, we presented a Web Services-based authorization framework to enforce organization-wide RBAC policies across various (Web Service) applications. Due to the fact that Web Services aim at integrating various applications of organizations and hence possibly expose security-critical functionality, it is desirable to enforce organizational rules on the Web Service level and to improve the security of an organization's business processes. In particular, we showed how our authorization framework integrates an authorization engine with the organization-wide applications by means of an interceptor, hence making the authorization engine as a central PDP. This way, the authorization logic is decoupled from the application logic. Furthermore, the authorization engine can be easily extended to support new types of authorization constraints (organizational rules) that are expressible in OCL. This way, the authorization engine can implement various kinds of authorization constraints, independent of the domain in question.

As part of future work, the authorization engine can be further extended to incorporate constraints on delegation. Last but not least, it would be interesting to integrate the authorization engine into a Workflow Management System (WfMS).

References

1. J. Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual, Second Edition. Reading, Mass., Addison Wesley Longman, 2004.
2. J. Warmer, A. Kleppe. The Object Constraint Language: Getting Your Models Ready for MDA, Addison-Wesley, 2003.
3. R. Sandhu, E. Coyne, H. Feinstein, C. Youman. Role-based access control models, IEEE Computer, vol. 29, no. 2, pp. 38–47, Feb. 1996.
4. American National Standards Institute Inc. Role Based Access Control, ANSI-INCITS 359-2004, 2004.
5. D.F. Ferraiolo, D.R. Kuhn, R. Chandramouli, Role-based access control, Artec House, Boston, 2003.
6. M. Nyanchama, S. Osborn. The graph model and conflicts of interest. ACM Trans. Inf.Syst. Sec. 2, 1, 1999.
7. G.-J. Ahn. The RCL 2000 language for specifying role-based authorization constraints, Ph.D. dissertation, George Mason University, Fairfax, Virginia, 1999.
8. V. D. Gligor, S. I. Gavrila, D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In 1998 IEEE Symposium on Security and Privacy, May 1998, pp. 172–185.
9. R. Simon, M. Zurko. Separation of duty in role-based environments, In 10th IEEE Computer Security Foundations Workshop (CSFW '97), June 1997, pp. 183–194.
10. K. Sohr, M. Drouineaud, G.-J. Ahn. Formal Specification of Role-based Security Policies for Clinical Information Systems, Santa Fe, New Mexico, in Proc. of the 20th ACM Symposium on Applied Computing, 2005.
11. J. Joshi, E. Bertino, U. Latif, A. Ghafoor. A generalized temporal role-based access control model. IEEE Trans. Knowl. Data Eng., vol. 17, no. 1, pp. 4–23, 2005.
12. E. Bertino, E. Ferrari, V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. ACM Trans. Inf. Syst. Secur., vol. 2, no. 1, pp. 65–104, 1999.

13. L. Zhang, G.-J. Ahn, B.-T. Chu. A role-based delegation framework for healthcare information systems, Proceedings of the 7th ACM Symposium on Access Control Models and Technologies, June 03-04, 2002, Monterey, California, USA.
14. American National Standards Institute Inc., Role Based Access Control, 2004, ANSI-INCITS 359-2004.
15. OASIS. eXtensible Access Control Markup Language (XACML), Vers. 2.0, February 2005.
16. M. Richters. A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis. Universität Bremen. Logos-Verlag, Berlin, BISS Monographs, No. 14. 2002.
17. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
18. T. Lodderstedt, D. Basin, J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security, UML, 5th International Conference. Vol. 2460. Dresden, Germany, pp.426-441, 2002.
19. I. Ray, N. Li, R. France, D.-K. Kim. Using UML to visualize role-based access control constraints. In Proc. of the 9th ACM Symposium on Access Control Models and Technologies, pp. 115–124, USA, 2004.
20. A. Anderson. Core and hierarchical role based access control (RBAC) profile of XACML v2.0, OASIS Standard, 2005.
21. D. Ferraiolo, D. Gilbert, N. Lynch. An examination of federal and commercial access control policy needs, in Proc. of the NIST-NCSC Nat. (U.S.) Comp. Security Conference, 1993, pp. 107–116.
22. K. Sohr, M. Drouineaud, G.-J. Ahn, M. Gogolla. Analyzing and Managing Role-Based Access Control Policies, IEEE Trans. Knowl. Data Eng., vol. 20., no 7, 2008.
23. C. K. Georgiadis, I. Mavridis, G. Pangalos, R. K. Thomas. Flexible team-based access control using contexts. In Proc. of the 6th ACM symposium on Access control models and technologies, p.21-27, May 2001, Chantilly, USA.
24. D.E. Denning. A lattice model of secure information flow, Communications of the ACM, vol. 19, no. 5, pp. 236-243, 1976.
25. G. Stampe. Personal Communication, 2007.
26. R.S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In Proc. 4th European Symposium on Research in Computer Security, 1996.
27. W3C: XML-Signature Syntax and Processing, 2002. Available at http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/
28. M. Brandner, M. Craes, F. Oellermann, O. Zimmermann. Web services-oriented architecture in production in the finance industry. Informatik Spektrum, vol. 27, no 2, pp. 136-145, 2004.
29. T. Priebe, W. Dobmeier, B. Muschall, G. Pernul. ABAC - Ein Referenzmodell für attributbasierte Zugriffskontrolle, Sicherheit 2005, pp. 285-296.
30. M. Zurko, R. Simon, T. Sanfilippo. A user-centered, modular authorization service built on an RBAC foundation. In Proc. of the IEEE Symp. On Sec. and Priv., pp. 57–71, Oakland, 1999.
31. R. Bhatti, A. Ghafoor, E. Bertino, J. Joshi. X-GTRBAC: an XML-based policy specification framework and architecture for enterprise-wide access control. ACM TISSEC, 8(2):187–227, 2005.
32. J. Crampton. Specifying and enforcing constraints in role-based access control, in Proc. of the 8th ACM Symposium on Access Control Models and Technologies. New York: ACM Press, June 2–3 2003, pp. 43–50.