Electronic Communications of the EASST
Volume 15 (2008)

Proceedings of the
8th International Workshop on
OCL Concepts and Tools (OCL 2008)
at MoDELS 2008

Implementing Advanced RBAC Administration Functionality with USE

Tanveer Mustafa, Karsten Sohr, Duc-Hanh Dang, Michael Drouineaud and Stefan Kowski

18 Pages

Guest Editors: Jordi Cabot, Martin Gogolla, Pieter Van Gorp
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Implementing Advanced RBAC Administration Functionality with USE[*]

Tanveer Mustafa[1], Karsten Sohr[1], Duc-Hanh Dang[1], Michael Drouineaud[1] and Stefan Kowski[2]

[1] Technologie-Zentrum Informatik, Universität Bremen
Bibliothekstraße 1
28359 Bremen, Germany
{tanveer, sohr, handd, mdruid}@tzi.de

[2] Parks Informatik GmbH
Girardetstr. 2-38
45131 Essen, Germany
stefan.kowski@parks-informatik.de

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 particular, today often only basic RBAC concepts have found their way into commercial RBAC products; specifically, authorization constraints are not widely supported. In this paper, we present an RBAC administration tool that can enforce certain kinds of role-based authorization constraints such as separation of duty constraints. The authorization constraint functionality is based upon the OCL validation tool USE. We also describe our practical experience that we gained on integrating OCL functionality into a prototype of an RBAC administration tool that shall be extended to a product in the future.

Keywords: Authorization constraints, Object Constraint Language, Role-based access control

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, government agencies 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) [1, 2, 3] as an efficient and flexible model for controlling access to computer resources (such as files or data base tables) and enforcing the organizational policies. In the RBAC model, users acquire permissions on resources via roles, and not directly.

[*] This work was supported in part by the German Research Foundation (DFG) under the grant SO 515/2-1 and by the German Federal Ministry of Education and Research under the grant FKZ01ISF19B (ORKA project).

As pointed out by Ferraiolo et al. [4], one of the main advantages of RBAC is that higher-level organizational rules can be implemented in a natural way. Specifically, advanced RBAC concepts like role-based authorization constraints and role hierarchies are a powerful means for laying out higher-level organizational rules [1]. Common types of authorization constraints are separation of duty (SoD) constraints [5, 6], cardinality constraints [1], and context constraints [7, 8].

Although the importance of authorization constraints[1] has long been pointed out [1, 9], advanced RBAC concepts are rarely well-supported in commercial RBAC products. In this paper, we demonstrate how authorization constraints can be implemented in a prototype of an RBAC administration tool. Specifically, we concentrate on static SoD constraints and role hierarchies. The prototype of the RBAC administration tool has been developed in the research and development project ORKA (Organizational Control Architecture) [21] comprised of various academic and industrial research partners (among the partners are SAP AG and Fraunhofer). In the future, it is envisioned to integrate this functionality into a real product made available by the Parks Informatik company [10].

Technically, the authorization constraints are implemented by employing functionality of the USE tool (UML-based Specification Environment), a validation tool for UML-/OCL-models [11]. With the help of this approach, authorization constraints are formulated as OCL invariants, and USE then checks whether the current system/security state satisfies the defined authorization constraints. The approach is based on our earlier works and is described in more detail elsewhere [12].

In this paper, we concentrate more on our practical experience employing a general-purpose OCL tool within the frameworks of a project with industrial partners. Specifically, we show that OCL tools such as USE can be employed in real-world industrial projects. However, we also demonstrate the problems we encountered by integrating the USE functionality with the RBAC administration tool.

The remainder of the paper is organized as follows: in Section 2 we provide a brief overview of related concepts and technologies. Section 3 presents our UML/OCL model of RBAC. In Section 4, we describe our implementation of authorization constraints with the help of the USE tool. We also describe our experience on employing USE in an industrial project. An overview of related work is given in Section 5. We outline our conclusions and future work in Section 6.

2 Related Concepts and Technologies

In this section, we first describe the RBAC concepts with the focus of authorization constraints. Thereafter, we explain the main functionality of USE.

2.1 RBAC and Authorization Constraints

RBAC [1, 2] 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 [2]. Here we give an

[1] In the following, we use the term "authorization constraint" instead of "role-based authorization constraint" for the sake of simplicity.

overview of the components of RBAC96, a widely used RBAC model introduced by Sandhu et al. [1]:

- 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 principal 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 [5, 6]; cardinality constraints [1]; context constraints [7, 8].

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.

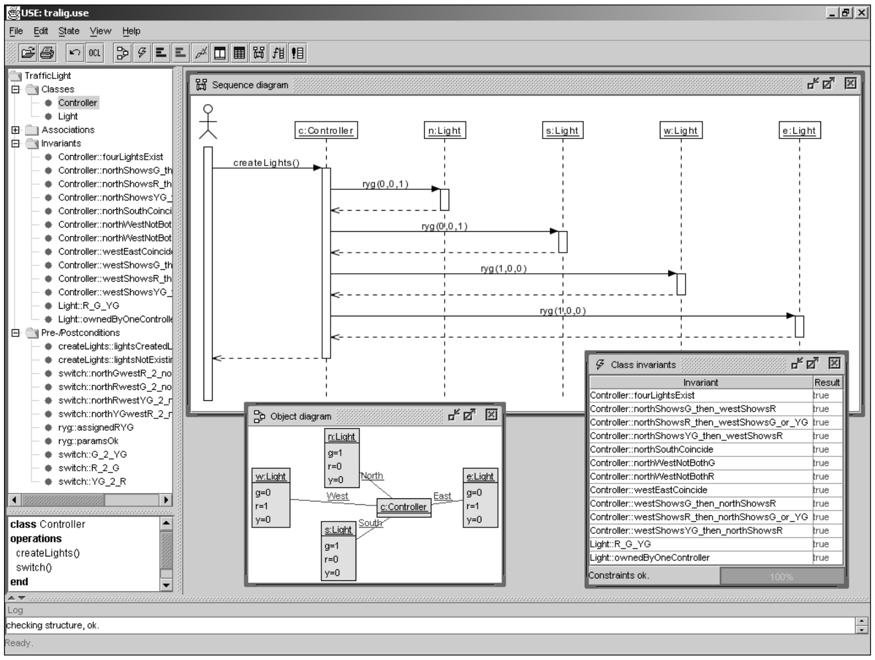
2.2 The USE tool

USE allows the software modeller to validate UML and OCL descriptions and 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 in the design level in an implementation-independent manner. This approach takes advantage of descriptive design level specifications by expressing properties concisely and in a more abstract way. Such properties are given by invariants and pre- and postconditions, and these are checked by the USE tool against the generated snapshots, i.e., object diagrams and operation calls given by sequence diagrams, which the developer provides. These abstract design level tests are expected to be also used later in the implementation phase.

The USE tool expects as an input a textual description of a model and its OCL constraints. After syntax checks, the model can be displayed by the graphical user interface provided by

USE. In particular, USE makes available a project browser which displays all the classes, associations, invariants, and pre- and post-conditions of the current model.



Invariant	Result
Controller::fourLightsExist	true
Controller::northShowsG_then_westShowsR	true
Controller::northShowsR_then_westShowsG_or_YG	true
Controller::northShowsYG_then_westShowsR	true
Controller::northSouthCoincide	true
Controller::northWestNotBothG	true
Controller::northWestNotBothR	true
Controller::westEastCoincide	true
Controller::westShowsG_then_northShowsR	true
Controller::westShowsR_then_northShowsG_or_YG	true
Controller::westShowsYG_then_northShowsR	true
Light::R_G_YG	true
Light::ownedByOneController	true

Constraints ok. 100%

```
class Controller
operations
  createLights()
  switch()
end
```

Log

```
checking structure, ok.
```

Ready.

Figure 1. Screenshot of the USE tool.

Figure 1 shows a USE screenshot 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 class invariant window and the validity of the pre- and post-conditions in the sequence diagram window.

3 Specifying RBAC in UML and OCL

Subsequently, we demonstrate how RBAC including authorization constraints can be specified in UML and OCL. Specifically, the RBAC element sets and relations are modeled in textual UML (which is defined within the USE tool), 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 RBAC policies[2]. The last point is discussed in the following section in more detail.

```
model RBAC

--classes

class Role
attributes
id:String
end

class User
attributes
id:String
end

class Permission
attributes
op:Operation
o:Object
end

class Object
attributes
id:String
end

class Operation
attributes
id:String
end

class Session
attributes
id: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

-- Simple Static SoD
context User inv SimpleSSoD:
let
  Clerk:Role=Role.allInstances->any(id='Clerk'),
  Supervisor:Role=Role.allInstances->any(id='Supervisor'),
  CR:Set(Role)=Set{Clerk, Supervisor}
in
  self.role_->intersection(CR)->size()< CR->size()

-- Simple Permission-Based Static SoD
context Role inv SimplePSSoD:
let
  loan:Object=Object.allInstances->any(id='loan'),
  prepare:Operation=Operation.allInstances->any
    (id='prepare'),
  approve:Operation=Operation.allInstances->any
    (id='approve'),
  approve_loan:Permission=Permission.allInstances->any
    (op=approve and o=loan),
  prepare_loan:Permission=Permission.allInstances->any
    (op=prepare and o=loan),
  cp: Set(Permission)=Set{prepare_loan, approve_loan}
in
  cp->intersection(self.permission)->size() < cp->size()

-- further authorization constraints …
```

Figure 2. USE specification of an RBAC policy.

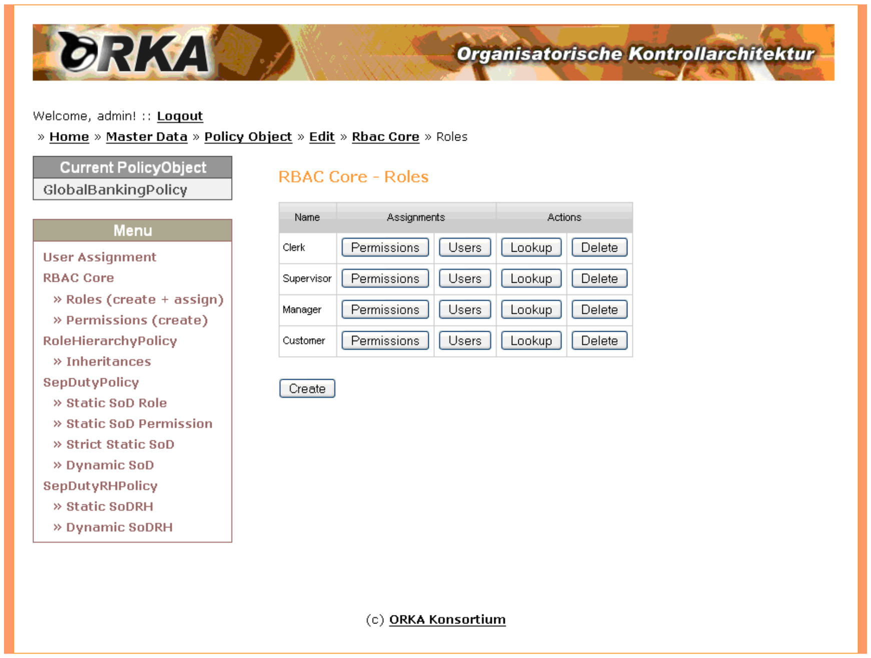
In Figure 2, we show a simple RBAC policy, which is represented as a USE specification. The USE specification consists of two parts. In the first part, the RBAC-related classes and association definitions are formulated in textual UML. This part is a generic encoding of RBAC. The second part then contains the domain-specific authorization constraints formulated in OCL. Specifically, we here define two constraints. The first is a Simple Static SoD (SimpleSSoD) constraint between two roles "Cashier" and "Cashier Supervisor", i.e., a user must not be assigned to both roles. The second constraint is of type Simple Permission-Based Static SoD (SimplePSSoD) stating that conflicting permissions cannot have a common role. Otherwise, the role in question would not be useful or even introduce a security hole. Both constraints are later used to explain our RBAC administration tool.

The RBAC policy depicted in Figure 2 is only meant for didactic purposes; it 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 addition, a lot of (mostly more complex) SoD constraints as those defined in [5, 6] can be specified in OCL.

[2] At minimum, an RBAC policy is comprised of users, roles, permissions, role hierarchies, user and permission assignment relations, as well as various constraints on those relations such as authorization and integrity constraints (cf. [12]).

4 Integrating USE Functionality into an RBAC Administration Tool

In previous works [12, 21], we demonstrated how to implement an authorization software with the help of the OCL validation tool USE. This approach has several advantages.



Welcome, admin! :: Logout

» Home » Master Data » Policy Object » Edit » Rbac Core » Roles

Current PolicyObject: GlobalBankingPolicy

RBAC Core - Roles

Name	Assignments		Actions	
Clerk	Permissions	Users	Lookup	Delete
Supervisor	Permissions	Users	Lookup	Delete
Manager	Permissions	Users	Lookup	Delete
Customer	Permissions	Users	Lookup	Delete

(c) ORKA Konsortium

Figure 3. ORKA-Admin tool.

First, one can utilize the benefits of the light-weight formalism OCL. Hence, a security officer can specify access control policies (i.e., sets of authorization constraints) in a declarative way. Thereafter, she can employ USE to validate this access control policy, for example, to detect missing or conflicting constraints under certain circumstances [12]. Last but not least, one can employ the USE functionality directly to implement/enforce the authorization constraints. Due to the fact that we use a general-purpose validation tool for OCL constraints new authorization constraint types can easily be added to the system. For example, if the access control policy must support cardinality constraints, one only has to specify (a template) for that new constraint type in OCL, and the authorization software can enforce the authorization constraint type.

In the following, we describe in more detail how the USE functionality is integrated with the RBAC administration tool made available in the ORKA project.

4.1 The RBAC Administration Tool

To reduce the complexity of security management an administrative interface is necessary to support an administrator to define, manage and analyze security policies and to trigger policy validation to detect inconsistencies and conflicts that may be violating underlying constraints.

Therefore, the ORKA-Admin tool, an RBAC policy administration tool, is being developed as part of the ORKA project.

In Figure 3, a screenshot of the ORKA-Admin tool is shown. The tool provides functionality for creating and managing RBAC policies. At the core, it supports standard RBAC administrative functions, such as creating users, roles, permissions, role hierarchies, assignment relations, and defining authorization constraints. While authorization constraints play a crucial role in enforcing organizational rules, they must be satisfied throughout the administration process. We take this fact into consideration by integrating USE validation functionality into the ORKA-Admin tool. The details are given later in Section 4.2. The USE validation primarily checks whether an RBAC policy satisfies the defined authorization constraints.

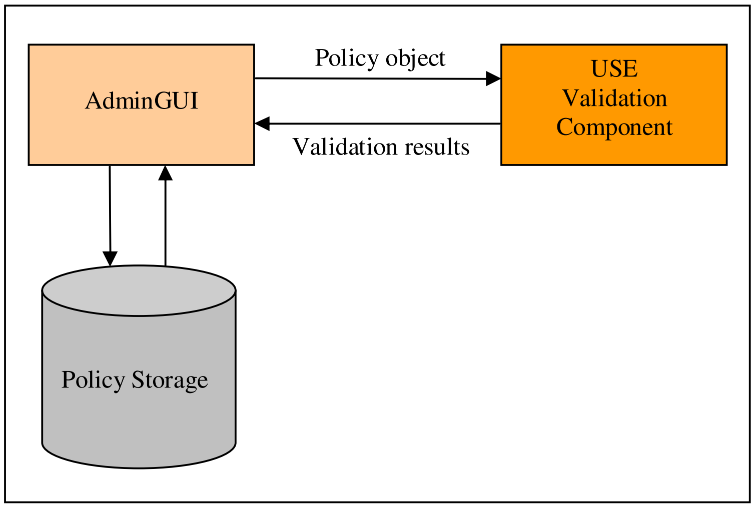


Figure 4. Components of the ORKA-Admin tool.

There are two types of USE validation that can be triggered from within the ORKA-Admin tool. First, the full validation of an RBAC policy, that is, an administrator can explicitly validate a *complete* RBAC policy. All possible conflicts that are detected by the validation process are reported back to the administrator in a user friendly manner. Second, we have an implicit operation-specific validation, that is, for each administrative operation, such as assigning permission(s) to a role, the USE validation is triggered automatically which checks only those conflicts that are caused by or are specific to the administrative operation in question.

Within the ORKA-Admin environment, the RBAC polices are usually analyzed, modified and validated as the working versions. Once the policies are validated, they can be deployed as production versions.

4.2 Architectural Overview

In this section, we provide more details regarding the components of the ORKA-Admin tool, specifically focusing on how the USE validation functionality is integrated. In Figure 4, an overview of the components of ORKA-Admin tool is given. The *AdminGUI* is a central place for all administrative activities. Internally within the ORKA-Admin environment, an RBAC policy is referred as *policy object* or *ORKA policy object*. It could also be called simply *ORKA policy*. The policy objects are saved into and retrieved from the central *Policy Storage* as XML documents, such as shown in Figure 5. The policy objects are validated automatically or explicitly on the behalf of the policy administrator by means of the *USEValidationComponent*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<policy_object>
  <policy_object_modules>
    <module_rbac_core_policy>
      <users>
        <user user_id="Jennifer" />
        <user user_id="Smith" />
        <user user_id="Suzanne" />
      </users>
      <roles>
        <role role_id="Clerk" />
        <role role_id="Supervisor" />
        <role role_id="Manager" />
        <role role_id="Customer" />
      </roles>
      <permissions>
        <permission permission_id="approve_loan">
          <operation operation_id="approve" />
          <object object_id="Loan" />
        </permission>
        <permission permission_id="prepare_loan">
          <operation operation_id="approve" />
          <object object_id="Loan" />
        </permission>
        <permission permission_id="query_customer_data">
          <operation operation_id="query" />
          <object object_id="CustomerData" />
        </permission>
      </permissions>
      <user_assignments>
        <user_assignment user_id="Jennifer" role_id="Manager" />
        <user_assignment user_id="Suzanne" role_id="Supervisor" />
      </user_assignments>
      <permission_assignments>
        <permission_assignment permission_id="approve_loan" role_id="Supervisor" />
        <permission_assignment permission_id="approve_loan" role_id="Manager" />
        <permission_assignment permission_id="query_customer_data" role_id="Clerk" />
        <permission_assignment permission_id="prepare_loan" role_id="Clerk" />
      </permission_assignments>
    </module_rbac_core_policy>
    <module_sep_duty_policy>
      <simple_static_separation_of_duty>
        <critical_role_sets>
          <critical_role_set cardinality="1">
            <critical_roles>
              <critical_role role_id="Clerk" />
              <critical_role role_id="Supervisor" />
            </critical_roles>
          </critical_role_set>
        </critical_role_sets>
      </simple_static_separation_of_duty>
      <static_separation_of_duty_attached_to_permissions>
        <critical_permission_sets>
          <critical_permission_set cardinality="1">
            <critical_permissions>
              <critical_permission permission_id="prepare_loan" />
              <critical_permission permission_id="approve_loan" />
            </critical_permissions>
          </critical_permission_set>
        </critical_permission_sets>
      </static_separation_of_duty_attached_to_permissions>
    </module_sep_duty_policy>
  </policy_object_modules>
</policy_object>
```

Figure 5. Fragment of a banking policy object.

The *AdminGUI* is a J2EE-based Seam application whereas the *Policy Storage* is a MySQL database server. The *USEValidationComponent* is built around the Java API made available by

USE. The *AdminGUI* and the *USEValidationComponent* communicate with each other through a common interface *PolicyValidatorInterface*.

4.2.1 Policy Representation Format

As indicated earlier, the ORKA-Admin tool internally uses XML to compose and store ORKA policies. An ORKA policy object in XML is the container for all policy rules of a particular application domain. There may be different policy objects for the various application domains. Each policy object is specified within a single XML file, which contains all policy rules. However, each policy object must conform to a central DTD (document type definition) which defines the syntax of the policy. That means the DTD provides a framework for the definition of syntactically correct policies in XML. Therefore, each policy object created or modified within the ORKA-Admin tool is validated against the central DTD.

In Figure 5, a fragment of a banking policy object is presented, which is created and exported from the ORKA-Admin tool. This policy object is only for didactic purposes, which by no means is a complete policy object that can be created, managed and validated (USE validation) by the tool. The policy object format allows specifying users, roles, permissions, role hierarchies, assignments relations and specifically various authorization constraints. For example all types of separation of duty constraints are specified within the module[3] `<module_sep_duty_policy>`. For Simple Static SoD (SimpleSSoD) and Strict Static SoD (StrictSSoD), we have an element `<critical_role_sets>` which holds all the `<critical_role_set>` elements of the particular type. A `<critical_role_set>` element contains the element `<critical_roles>` which includes the critical roles as `<critical_role>` elements. Additionally, the `<critical_role_set>` has a mandatory attribute `cardinality` which specifies the cardinality of the respective role set. For instance, in Figure 5 (lines 44-49), a constraint of type SimpleSSoD is specified, which informally means that no user is allowed to be assigned to the critical role set comprised of the `Clerk` and `Supervisor` roles. Similarly, lines 54-59 of Figure 5 show a Simple Permission-Based Static SoD (SimplePSSoD) constraint, which states that the critical permission set comprised of `prepare_loan` and `approve_loan` cannot be assigned to the same role.

More complex authorization constraints, including role hierarchies and associated constraints such as partial order constraints (e.g., anti-symmetry and transitivity) can be created by using the ORKA-Admin interface, which are internally stored in the respective policy object.

4.2.2 USE Validation

Although the ORKA-Admin tool implements a user interface to create and manage policy objects that are internally stored in the XML format, a critical requirement is to validate the policy objects, specifically, whether the policy objects satisfy all the defined authorization constraints. The validation must be carried out on the policy objects before they are deployed as production versions. The *USEValidationComponent* is developed around the Java API provided by USE and integrated into the ORKA-Admin tool, which facilitates validating the

[3] Within the ORKA project several modules containing exact specification(s) of authorization constraint types are provided.

policy objects and sending back immediate feedback to the *AdminGUI* to reduce administrative mistakes.

As pointed out before, the OKRA-Admin tool supports an implicit operation-specific validation as well as an explicit full validation that can be triggered by the administrator at any time. In case of full validation, the *AdminGUI* sends a complete policy object as an XML string to the *USEValidationComponent.* However, in case of operation-specific validation, the operation-specific parameters are also sent along with the policy object.

On receiving the validation request from the *AdminGUI*, the *USEValidationComponent* carries out the following steps:

1. It initializes an internal USE model comprised of various classes and associations, such as those shown in Figure 2. From the USE model, it also instantiates a USE system representing a single ORKA policy object. This USE system allows one to create, preserve and manipulate unique objects of type `Role`, `User` and `Permission,` as well as the association links such as `UA`, `PA` and `RH` as shown in Figure 2.
2. The authorization constraints are read from the policy object, transformed into equivalent OCL invariants with respect to the specifications given in policy object modules, and uniquely created into the USE model. For example, from Figure 5 the SimpleSSoD (lines 44-49) and SimplePSSoD (lines 54-59) constraints will be translated as SimpleSSoD and SimplePSSoD OCL invariants, respectively, as shown in Figure 2.
3. The concrete roles, users and permissions are read from the policy object, and corresponding unique objects of type `Role`, `User` and `Permission` are created in the current USE system state. For example, a unique user object, say, *user_clerk* of type `User` will be created for the user `<user user_id="Jennifer" />` as specified in the policy object in Figure 5. The `id` of the object *user_clerk* will be set to “Jennifer”.
4. The role hierarchy, user assignment, and permission assignment relations are read from the policy object and are created as respective association links in the current system state. While creating role hierarchy and assignment relations, the reflexive transitive closure is calculated. For instance, the USE system state contains the role hierarchy with all possible edges computed by the transitive closure algorithm.
5. Finally, the USE system evaluates the current system state with respect to the existing invariants. If an explicit (full) validation is trigged by the ORKA-Admin tool, then all existing invariants are checked. In case of operation-specific validation, the invariants to be checked are selected on the basis of the administrative operation being invoked by the ORKA-Admin tool. For all violated invariants, the *USEValidationComponent* analyzes invariant types and generates specific messages to be sent back to the *AdminGUI.* Each message is formatted as an XML file and sent back to the *AdminGUI* as XML string, such as shown in Figure 6. OCL queries are applied directly on the USE system state to retrieve specific information wherever required.

In the following section, we describe in more detail how various authorization constraints are implemented by the *USEValidationComponent.* Thereafter, more details regarding message generation and OCL queries are given in Section 4.4.

4.3 Implementing Static Authorization Constraints and Role Hierarchy Relations

The *USEValidationComponent* of the ORKA-Admin tool implements various constraints that can be specified using the tool interface. Specifically, we implemented partial order constraints (e.g., anti-symmetry) and various static SoD constraints such as SimpleSSoD, StrictSSoD, SimplePSSoD and Strict Permission-Based SSoD (StrictPSSoD).

The *USEValidationComponent* follows a template mechanism to implement the aforementioned constraint types. To describe it simply, a **constraint template class** (Java class) is defined for each type of authorization constraint. For instance, the `SimplePSSoDConstraint` template class implements constraints of type SimplePSSoD such as shown in Figure 5. The *USEValidationComponent* will therefore create a new instance of the `SimplePSSoDConstraint` template class for each SimplePSSoD constraint that is read from the policy object. These template instances which are capable of producing corresponding OCL invariants are preserved throughout the life cycle of the USE system. The OCL invariants are then added to the USE model accordingly.

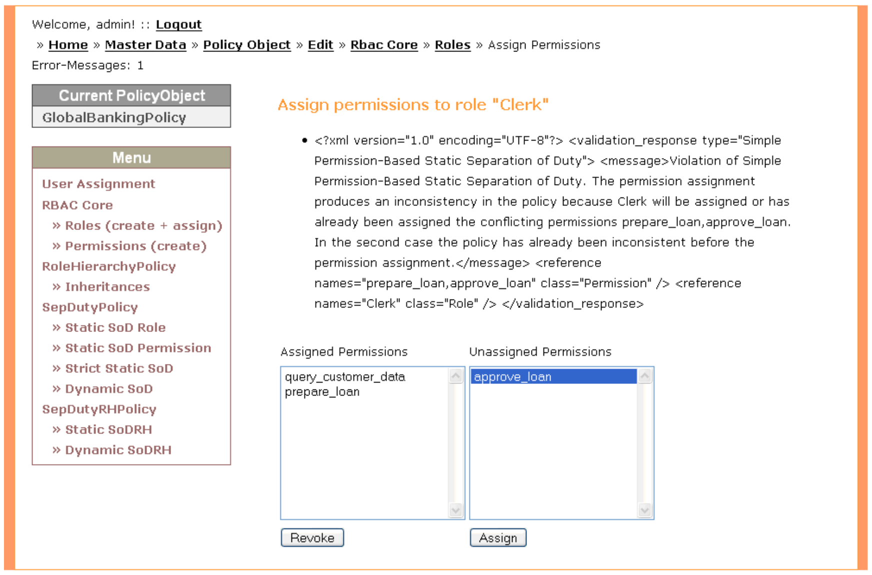
As an example of how template classes are instantiated for specific constraint types and what information they hold, consider the policy object shown in Figure 5, specifically, the SimplePSSoD constraint specified between lines 54-59. Within the *USEValidationComponent* an instance of the template class `SimplePSSoDConstraint` is created for the SimplePSSoD constraint, which at least holds the critical permission set. This instance can then be manipulated, for example, to produce the corresponding OCL invariant. In the current scenario, it will produce the following OCL invariant:

```
context Role inv simplepssod_uniqueID:
let
  loan:Object=Object.allInstances->any(id='loan'),
  prepare:Operation=Operation.allInstances->any
    (id='prepare'),
  approve:Operation=Operation.allInstances->any
    (id='approve'),
  approve_loan:Permission=Permission.allInstances->any
    (op=approve and o=loan),
  prepare_loan:Permission=Permission.allInstances->any
    (op=prepare and o=loan),
  cp: Set(Permission)=Set{prepare_loan, approve_loan}
in
  cp->intersection(self.permission)->size()< cp->size()
```

In fact, within the *USEValidationComponent*, an instance of the template class `SimplePSSoDConstraint` will be created for each set of conflicting permissions specified in the policy object. The template class instances are used to produce corresponding OCL invariants, which have unique names within the USE model. When we create the OCL invariant in the USE model, the invariant is mapped to the template instance to which it belongs. This mapping is necessary for the later use while analyzing the violation of specific invariants and producing corresponding error messages.

4.4 Generating Error Messages

The template classes described in the previous section are not bound to produce only OCL invariants. The template classes also hold a set of preformatted OCL queries that can be directly applied on the USE system state to retrieve specific information of the current USE system state. In particular, each template class is capable of producing specific warning/error messages when the OCL invariant, it refers to in the USE system state, is violated. While generating a specific error message, the template instance primarily uses the information it already holds. For example, an instance of class `SimplePSSoDConstraint` holds a critical permission set. In addition, it can apply OCL queries on the current USE system state to retrieve further information, if required.



Error-Messages: 1

GlobalBankingPolicy

Assign permissions to role "Clerk"

- <?xml version="1.0" encoding="UTF-8"?> <validation_response type="Simple Permission-Based Static Separation of Duty"> <message>Violation of Simple Permission-Based Static Separation of Duty. The permission assignment produces an inconsistency in the policy because Clerk will be assigned or has already been assigned the conflicting permissions prepare_loan,approve_loan. In the second case the policy has already been inconsistent before the permission assignment.</message> <reference names="prepare_loan,approve_loan" class="Permission" /> <reference names="Clerk" class="Role" /> </validation_response>

Assigned Permissions	Unassigned Permissions
query_customer_data	approve_loan
prepare_loan	

Figure 6. The ORKA-Admin tool showing an operation-specific USE validation result.

Here we present two examples which describe the USE validation results. For the first example, an operation-specific USE validation scenario is depicted in Figure 6, which is based upon the policy object shown in Figure 5. In this case when an administrator tries to assign permission `approve_loan` to the role `Clerk`, then the operation-specific validation is automatically triggered. As a result, the policy object and operation-specific information, such as the operation name (*AssignPermissionToRole*) and attribute list, that is, the role `Clerk` and the permission `approve_loan`, is sent to the *USEValidationComponent* to check whether the current operation violates defined authorization constraint(s). The *USEValidationComponent* carries out different steps to initialize the USE model and the USE system as discussed earlier. In this specific case, while creating the permission assignment relations in the system, the permission `approve_loan` will also be assigned to the `Role` object whose `id` is set to “Clerk”. The same role object has already been assigned a permission `prepare_loan` which is based upon the information retrieved from the policy object. Within the USE system state, there now would be two `Permission` objects with the ids “prepare_loan” and “approve_loan”, and which are assigned to a Role object with the id “Clerk”. Hence, the Permission-Based Static SoD constraint is violated.

Further, apart from other invariants, there would be an invariant such as simplepssod_uniqueID discussed in Section 4.2, which would always be mapped to the corresponding instance of the template class SimplePSSoDConstraint. When USE evaluates invariants in the current system state, the invariant simplepssod_uniqueID will be evaluated to false because within USE system state a Role object is assigned, at least, two Permission objects referring to the critical permissions “prepare_loan” and “approve_loan”.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<policy_object>
  <policy_object_modules>
    <module_rbac_core_policy>
      <users>
        <user user_id="Smith" />
      </users>
      <roles>
        <role role_id="Clerk" />
        <role role_id="Supervisor" />
      </roles>
      <user_assignments>
        <user_assignment user_id="Smith" role_id="Clerk" />
        <user_assignment user_id="Smith" role_id="Supervisor" />
      </user_assignments>
    </module_rbac_core_policy>
    <module_sep_duty_policy>
      <simple_static_separation_of_duty>
        <critical_role_sets>
          <critical_role_set cardinality="1">
            <critical_roles>
              <critical_role role_id="Clerk" />
              <critical_role role_id="Supervisor" />
            </critical_roles>
          </critical_role_set>
        </critical_role_sets>
      </simple_static_separation_of_duty>
    </module_sep_duty_policy>
  </policy_object_modules>
   </policy_object>
```

Figure 7. Fragment of an example policy object.

The USE system produces its own internal evaluation log for each invariant which is evaluated to false. The evaluation log can be analyzed to find the cause of the failure in detail. However, for the ORKA-Admin tool we need to produce specific messages for the violated constraints that are useful for an administrator. Therefore, in the scenario being discussed here, the *USEValidationComponent* would acquire the preserved instance of SimplePSSoDConstraint that is mapped to the violated invariant simplepssod_uniqueID. This way, the SimplePSSoDConstraint instance will generate an XML based message as shown in Figure 6. The SimplePSSoDConstraint instance does not execute any OCL query on the USE system state for any more information because it already holds the necessary information that is required to produce the message. In the current version of the ORKA-Admin tool the error messages displayed are complete XML strings. However, error messages are supposed to be further parsed to create hyperlinks to different elements such as users, roles and permissions to help an administrator to navigate to the linked elements.

For the full validation case, we consider the example policy object shown in Figure 7. To keep it simple, we are considering only a small fragment of the policy object which contains only one conflict. In case of full validation, only the policy object is sent to the

USEValidationComponent. During the process of creating invariants, an instance of the template class `SimpleSSoDConstraint` will be created for each constraint of type SimpleSSoD read from the policy object. In our example policy object there is only one authorization constraint specified between lines 20-25. The aforementioned `SimpleSSoDConstraint` instance will produce the following OCL invariant, which is then added to the USE model:

```
context User inv simplessod_uniqueID:
let
  Clerk:Role=Role.allInstances->any
    (id='Clerk'),
  Supervisor:Role=Role.allInstances>any
    (id='Supervisor'),
  CR:Set(Role)=Set{Clerk, Supervisor}
in
  self.role_->intersection(CR)->size()< CR->size()
```

While checking invariants in the USE system state, the invariant `simplessod_uniqueID` is evaluated to false. The `SimpleSSoDConstraint` instance corresponding to the invariant `simplessod_uniqueID` will therefore generate an XML message as shown in Figure 8. In the XML message, we also need to indicate all those users who are assigned to the critical role set comprised of the `Clerk` and `Supervisor` roles. In other words, we need to indicate all those users who are violating the `simplessod_uniqueID` invariant. However, the `SimpleSSoDConstraint` instance only holds the critical role set, and is not aware of the users that are assigned to the critical role set. While OCL queries play an important role in retrieving specific information from the USE system state, some of the template classes contain preformatted OCL queries. These queries acquire concrete values from the corresponding template class instances. For example, in the above case, the `SimpleSSoDConstraint` instance contains the critical role set and it will dynamically build the following concrete OCL query, which is then executed on the USE system state:

```
let
  Clerk:Role=Role.allInstances->any(id ='Clerk'),
  Supervisor:Role=Role.allInstances->any
    (id ='Supervisor'),
  cr : Set(Role)=Set{Clerk,Supervisor}
in
  User.allInstances->reject(u| u.role_->intersection
    (cr)->size()< cr-> size())->iterate(u:User;
    result:Set(String)=oclEmpty(Set(String))|
      result->union(Set{u.id}))
```

The query will return a set of users that are assigned to the critical role set. In our case, the resulting set would contain only one user named “Smith”.

There is also a way to automate the process of generating queries from the authorization constraints formulated as OCL invariants. For example, if you take a look at the SoD constraints given in Figure 2, you can see that they are of the form

```
context C inv:
let
 …
in
  condition
```

For the feedback of the USE system, we are interested in instances of the class `C` which are violating the condition. Thus, we can obtain a corresponding query of the following form:

```
let
 …
in
  C.allInstances->reject(c| condition)
```

Note that all `self` expressions must be replaced by the instance `c` because we do not have a context here.

4.5 Lesson Learned

We demonstrated that it is possible to integrate USE functionality with an industrial RBAC administration tool. The strength of this approach lies in its flexibility, i.e., various forms of static SoD can be implemented and new forms can be added relatively easily. Due to the fact that we always create a new USE system state to validate a policy object, this approach may slow down the RBAC administration task if the underlying policy object of larger size has to be validated automatically for each administrative operation. Therefore, an offline validation is also provided, that is, a policy object can be validated at once before the deployment.

The main work in this approach remained to produce understandable warning/error messages, i.e., to interpret the feedback from USE. For each type of authorization constraint, specially tailored messages must be constructed (cp. Section 4.4). However, we gave in Section 4.4 a scheme how to automate the process of retrieving feedback from USE.

Furthermore, there are other tasks that could be carried out with the help of USE. For example, one might want to check if administrative RBAC operations have unexpected side effects. For example, a permission might be revoked from a role `r`, and as an unexpected side effect, it might also be revoked from a role senior to `r`. Thus, the query functionality would be helpful to detect such effects. Due to the fact that only a few side effect checks have been considered in ORKA, it was decided not to utilize USE for that purpose, but implement such checks in an ad hoc fashion.

5 Related Work

There is a plethora of works in the context of embedding RBAC into UML/OCL such as [13, 14, 15, 16]. In addition, our results presented in this paper are based upon our earlier work [12]. There, we showed how to build an authorization engine by means of the USE functionality. In contrast, the focus of this paper lies more on integrating the USE functionality with an industrial RBAC administration tool.

As indicated above, the USE system is a general-purpose validation tool and can hence be employed for the other UML/OCL encodings of RBAC policies mentioned above. In particular, Basin et al. present a modeling language SecureUML for integrating the specification of access control into application models [13]. Extending their work, Basin et al. present a validation approach, which allows one to automatically analyze RBAC policies formulated in UML/OCL [17]. OCL queries on RBAC policies can be automatically evaluated, i.e., RBAC policies can be tested for non-trivial access control requirements. The theoretical foundations of queries are given through meta-modeling. In addition, a validation tool, called SecureMOVA, is made available for checking RBAC policies. Similarly, our RBAC admin tool could be extended with such a query functionality to check access control requirements (beyond static SoD properties).

RBAC functionality is also incorporated into many products such as operating systems, applications (e.g., clinical information systems, banking software), and databases. Specifically, enterprise administration tools such as DirXMetaRole from Siemens [18], or the Jupiter system from Beta Systems [19] support RBAC. However, most of these engines only implement basic RBAC concepts. If authorization constraints are supported at all, they are mostly limited to Simple Static SoD (which is also defined in the ANSI standard for RBAC [2]). Other types of authorization constraints are rarely implemented.

In addition, a comparison of our work with XACML is also worthwhile. XACML is an OASIS standard that supports the specification of authorization policies and related queries in a standardized, machine-readable way [22]. 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 [23]. UML/OCL, however, is a standard modelling approach that can be used to express the RBAC policies more abstractly 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 and time consuming.

6 Conclusions and Outlook

We demonstrated in this paper how to implement advanced administrative RBAC functionality by means of the USE tool. In particular, static authorization constraints such as Simple Static SoD and Permission-Based Static SoD have been implemented with the help of this approach. Other types of authorization constraints such as cardinality constraints can also be implemented. This way, the RBAC administration tool is extensible and helps to keep RBAC policies consistent with respect to defined authorization constraints. Implementing the static authorization constraints is comparatively easy with the USE tool. However, in case of authorization constraint violation(s) the essential requirement is to retrieve the relevant information from the USE system and to generate adequate error messages for the ORKA-Admin tool. Due to the fact that the RBAC administration tool is still being developed within the frameworks of a research project with industrial partners there is hope that OCL functionality will be used in security products in the future.

In addition, USE functionality can also be employed for implementing dynamic authorization constraints such as History-Based SoD [20]. This way, a policy decision point for workflow engines [21] can be realized based upon an OCL tool. This, however, remains future work. Other RBAC encodings such as SecureUML could also be implemented through our USE approach. Last but not least, our approach is not restricted to RBAC or IT security in general. It could also be applied to problems in other domains such as safety-critical systems.

References

1. R. Sandhu, E. Coyne, H. Feinstein, C. Youman. Role-based access control models, IEEE Computer, vol. 29, no. 2, pp. 38–47, Feb. 1996.
2. American National Standards Institute Inc. Role Based Access Control, ANSI-INCITS 359-2004, 2004.
3. D.F. Ferraiolo, D.R. Kuhn, R. Chandramouli, Role-based access control, Artec House, Boston, 2003.
4. 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.
5. G.-J. Ahn. The RCL 2000 language for specifying role-based authorization constraints, Ph.D. dissertation, George Mason University, Fairfax, Virginia, 1999.
6. 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.
7. 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.
8. 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.
9. T. Jaeger and J. Tidswell. Practical Safety in Flexible Access Control Models, ACM Trans. Information and System Security, vol. 4, no. 2, pp. 158-190, May 2001.
10. Parks Informatik. The Parks Security Manager, 2008 http://www.parks-informatik.de/de/product/psm/ParksSecurityManagement.html
11. M. Richters. A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis. Universität Bremen. Logos-Verlag, Berlin, BISS Monographs, No. 14. 2002.
12. 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.
13. 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.
14. 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.
15. T. Priebe, W. Dobmeier, B. Muschall, G. Pernul. ABAC - Ein Referenzmodell für attributbasierte Zugriffskontrolle, Sicherheit 2005, pp. 285-296.
16. Gail-Joon Ahn , Michael E. Shin, Role-Based Authorization Constraints Specification Using Object Constraint Language, Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, p.157-162, June 20-22, 2001.
17. David Basin and Manuel Clavel and Jürgen Doser and Marina Egea. Automated Analysis of Security-Design Models. In Information and Software Technology, 2008.
18. Siemens AG. DirXMetaRole Administration Guide.
19. Beta Systems Software AG. SAM Jupiter User Guide, 2008 http://ww2.betasystems.com/de/produkte/idm/produkte/sam_jupiter.html
20. 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.
21. Project ORKA. http://www.orka-projekt.de/index-en.htm
22. OASIS. eXtensible Access Control Markup Language (XACML), Vers. 2.0, February 2005.
23. A. Anderson. Core and hierarchical role based access control (RBAC) profile of XACML v2.0, OASIS Standard, 2005.