Comprehensive Two-Level Analysis of Role-Based Delegation and Revocation Policies with UML and OCL

Karsten Sohr, Mirco Kuhlmann, Martin Gogolla

Center for Computing Technologies (TZI), Universität Bremen

Hongxin Hu, Gail-Joon Ahn

Arizona State University

Abstract

Context. Role-based access control (RBAC) has become the de facto standard for access management in various large-scale organizations. Often role-based policies must implement organizational rules to satisfy compliance or authorization requirements, e.g., the principle of separation of duty (SoD). To provide business continuity, organizations should also support the delegation of access rights and roles, respectively. This, however, makes access control more complex and error-prone, in particular, when delegation concepts interplay with SoD rules.

Objective. A systematic way to specify and validate access control policies consisting of organizational rules such as SoD as well as delegation and revocation rules shall be developed. A domain-specific language for RBAC as well as delegation concepts shall be made available.

Method. In this paper, we present an approach to the precise specification and validation of role-based policies based on UML and OCL. We significantly extend our earlier work, which proposed a UML-based domain-specific language for RBAC, by supporting delegation and revocation concepts.

Result. We show the appropriateness of our approach by applying it to a banking application. In particular, we give three scenarios for validating the interplay between SoD rules and delegation/revocation.

Preprint submitted to Information and Software Technology *April 19, 2012*

Conclusion. To the best of our knowledge, this is the first attempt to formalize advanced RBAC concepts, such as history-based SoD as well as various delegation and revocation schemes, with UML and OCL. With the rich tool support of UML, we believe our work can be employed to validate and implement real-world role-based policies.

Keywords:
UML, OCL, RBAC, Delegation, Revocation

---

1. Introduction

Today, role-based access control (RBAC) is widely-used to simplify access management in various large-scale organizations, such as financial institutes, healthcare providers, and enterprises. In RBAC, users obtain access to business processes and resources through roles rather than directly through permissions. One important advantage of RBAC is that organizational rules, such as separation of duty (SoD), can be naturally specified and implemented. Proceeding this way, various kinds of role-based authorization constraints have been proposed in literature [1, 2, 3, 4, 5]. In contrast to common static SoD constraints (e.g., no user can be assigned to the roles cashier and cashier supervisor by an administrator), dynamic authorization constraints are more flexible and hence are needed in many organizations. A typical example of dynamic SoD is the rule “A check must not be prepared, verified and signed by the same clerk”. In this case, the access decision for signing the check depends on the actions that have been previously performed by the clerk. In literature, several types of dynamic authorization constraints have been discussed, such as history-based SoD and resource-based dynamic SoD[1] [2, 6].

Another advanced access control concept is role-based delegation and revocation [7, 8, 9, 10, 11]. Many organizations may often face situations in which employees need specific access rights in an ad-hoc fashion, without involving security administrators. For example, consider a situation in which an order must be timely confirmed, but the supervisor is not available due to work overload. In this case, the supervisor may need to partially delegate

[1] In the access control literature, this constraint is often referred to as “object-based dynamic SoD”. Due to the fact that the term “object” has another meaning in the context of UML, we use “resource” instead.

her supervisor role to another colleague to process the confirmation request. In a healthcare environment, physicians often need to consult a specialist, and hence, need to pass on patient data. In such a situation, the attending physician needs to delegate read access for the patient data to the specialist. Sometimes delegations should be revoked, e.g., on returning from vacation, a supervisor revokes privileges from her colleague who was member of the delegated role.

Role-based policies, i.e., the defined roles and authorization constraints, might become quite complex through the interplay between authorization constraints and delegation/revocation schemes. For example, role delegation may cause unexpected violations of a history-based SoD constraint since delegation itself might conflict with the delegatee's previous activities. For this reason, it is desirable to allow a security officer to systematically create and analyze the defined role-based policies.

In our earlier work, we introduced an approach to modeling role-based policies—including dynamic authorization constraints—with the Unified Modeling Language (UML) and the Object Constraint Language (OCL) [12]. In particular, we designed a domain-specific language (DSL), based on a UML metamodel using class diagrams and OCL constraints. Role-based concepts (including various kinds of authorization constraints) have been formalized at the level of the metamodel. The concrete policies of an organization can then be defined with the help of UML object diagrams. Proceeding this way, an administrator/security officer can use a more intuitive graphical UML-based DSL, while the technical details (e.g., the OCL constraints) can be hidden in the metamodel. In addition, using UML/OCL for policy specification allows an organization to utilize the rich tool support available for UML, such as CASE and UML validation tools, which are widely adopted by industry, in contrast to logic-based policy languages. Due to the fact that our RBAC DSL has its foundation in UML object diagrams, which is a basic diagram type, UML tools generally support our RBAC DSL. Approaches which utilize the UML profile mechanisms, e.g., SecureUML [13], have the drawback that it is unclear if and to which extent CASE tools handle this extension mechanism.

Having a UML-based formalization of role-based policies at hand, we showed how to employ the USE model validator [14] and the USE tool (UML-based Specification Environment) [15] to validate the metamodel as well as role-based policies. The validation approach with USE allows an administrator to define concrete test cases for RBAC policies as UML object diagrams.

A test case, for example, can be a situation (system state) in which a specific user can execute certain permissions which she should never obtain. If the policy allows this situation, this might be a hint that the policy is incorrect.

In this paper, we significantly extend our earlier works in supporting delegation and revocation concepts. Our work is based on RDM2000, a well-established delegation and revocation framework [7], which includes constrained delegation (i.e., constraints are imposed on delegation) as well as different revocation schemes as introduced by Hagström et al. [16]. Proceeding this way, our metamodel can encompass important access control features in combining advanced RBAC concepts (e.g., static and dynamic authorization constraints and role-hierarchies) with delegation/revocation schemes. This allows us to capture complex access control requirements demanded by real-world applications. We further show the applicability of our concepts with the help of a banking application defined by Chandramouli [17]. Last but not least, we employ USE and the USE model validator to validate role-based policies in order to detect subtle problems stemming from the combination of authorization constraints, role hierarchies, and delegation.

In summary, our main contribution is a UML-based RBAC DSL, which is expressive enough to support delegation and revocation schemes as well as advanced RBAC concepts, such as role hierarchies and history-based authorization constraints. The DSL is graphic-based and hence easy to understand in contrast to approaches which utilize logics for policy specification. In addition, a policy designer can employ CASE tools (including functionality for model driven development) to specify, analyze and implement the role-based policies.

The remainder of this paper is organized as follows. Section 2 gives an overview of the basic technologies and concepts used in this paper. We then recapitulate the RBAC metamodel in Section 3, whereas Section 4 discusses the validation of this metamodel. In Section 5, we describe the extensions of the metamodel to support delegation and revocation schemes and thereafter show how to apply our RBAC DSL to a banking application. After discussing related work in Section 7, we conclude and give an outlook in Section 8.

2. Background

In the following, we describe the background of our work including the basic techniques as we have done in [18] with the exception of delegation concepts which have not been covered. Having briefly described the main

concepts behind UML, OCL, DSLs, and the validation tool USE, we recapitulate access control concepts, such as RBAC, authorization constraints as well as delegation and revocation. In particular, we give an overview of the RDM2000 delegation model, which will be a central aspect of our current work.

If the reader is familiar with the concepts treated in this section, she might skip them and move on to Section 3.1.

2.1. Employed Modeling and Validation Approaches

2.1.1. Unified Modeling Language

The Unified Modeling Language (UML) [19, 20] represents a general-purpose visual modeling language in which we can specify, visualize, and document the components of software and hardware systems. It captures decisions and understanding about systems that are to be constructed. UML has become a standard modeling language in the field of software engineering and is increasingly used in hardware/software co-design.

Through different views and corresponding diagrams, UML permits the description of static, functional, and dynamic models [21]. In this paper, we concentrate on UML class and object diagrams. A class diagram 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, but also association classes which allow for adding further information to the relationships. Object diagrams visualize instances of the modeled system, i. e., class instances (objects), attribute instances (values) and instances of associations (links).

Figure 1 shows an example class and object diagram. The class diagram visualizes a small UML model consisting of the classes ‘Person’ which has the attributes ‘name’ and ‘age’ and ‘Company’ also containing an attribute ‘name’. Persons may be related through the binary reflexive association ‘Parenthood’. The association ends ‘parent’ and ‘child’ determine the roles a person can assume in a parenthood relationship. Persons can have jobs, as the association class ‘Job’ relates them with companies. The attribute of the association class holds the salary for each job. Since persons may have more than one job, the operation ‘salary()’ of class Person calculates the sum of all related salaries. Relationships between classes may be constrained by multiplicities. In our example, a person may have any number of children, but at most two parents. A company must have at least one employee.

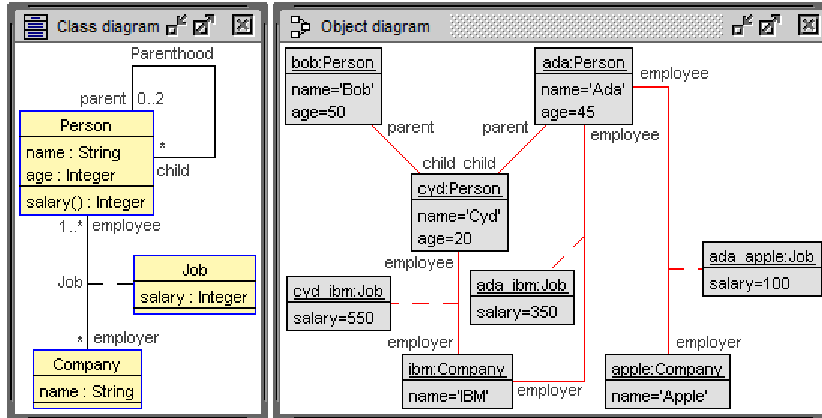

Figure 1: Example UML class and object diagram

The object diagram represents an example instance of the model including a family with jobs at two different companies. Ada, for example, is employed at IBM and Apple which pay individual salaries. Bob is unemployed.

2.1.2. Object Constraint Language

The Object Constraint Language (OCL) [22] is a declarative textual language that describes constraints on object-oriented models. It is an industrial standard for object-oriented analysis and design.

OCL expressions consist of OCL standard operations or user-defined OCL query operations. The built-in standard operations support calculations on the basic types Boolean (e. g., `and`, `or` and `implies`), Integer (e. g., `+`, `*` and `mod`), Real (e. g., `/`, and `round`), as well as on collection types, i. e., sets, bags (multiset), ordered sets and sequences. Beside the usual collection type operations (e. g., `union`, `size` and `includes`) several operations enable iteration over the members of a collection such as `forAll`, `exists`, `iterate`, and `select`. The most important features of OCL are navigation and attribute access, which connect an OCL expression with the values in a concrete model instance. By definition, OCL constraints can restrict the static aspects of a UML model through invariants. Dynamic aspects with respect to user-defined class operations and their expected execution results are addressed through pre- and postconditions. In this paper, we break this distinction by explicitly integrating the dynamic problems into our RBAC metamodel enabling our invariants to enforce temporal properties.

OCL invariants are related to a context class; i. e., the boolean expression for an invariant is evaluated for each instance of this class. If the expression evaluates to false in the context of at least one object, the invariant is violated, indicating an invalid model instance. The reserved word 'self' is used to refer to the contextual instance. We extended our example UML model presented in Figure 1 by the two simple invariants which are named 'minimumWage' and 'minumumAge'.

```
context Person inv minimumWage:
  self.employer->notEmpty() implies self.salary() >= 500
```

The first invariant describes a logical implication whose premise checks whether the considered Person object has at least one employer. The subexpression `self.employer` is a navigation from an object (self) via the association end employee to a set of linked Company objects. The collection operation `notEmpty` evaluates to true if the source collection includes at least one element. We implemented the operation salary() as an OCL query operation which calculates the total income of a person without side-effects (i. e., without changing the model instance).

```
Person::salary() :  Integer = self.job.salary->sum()
```

After navigating from a person to her jobs, the attribute salary of each Job object is accessed and all corresponding values are collected in a bag. In the end, the sum of all elements of the bag is returned. Consequently, the invariant demands each working person to earn at least 500 units.

The second invariant makes use of the operation `forAll`, which iterates over each person who is employed in the considered company, and evaluates the boolean body expression `p.age >= 16`.

```
context Company inv minimumAge:
  self.employee->forAll(p | p.age >= 16)
```

2.1.3. UML-Based Specification Environment

The UML-based Specification Environment (USE) supports the validation of UML and OCL descriptions. USE is the only OCL tool enabling interactive monitoring of OCL invariants and pre- and postconditions, as well as automatic generation of non-trivial model instances. The central idea

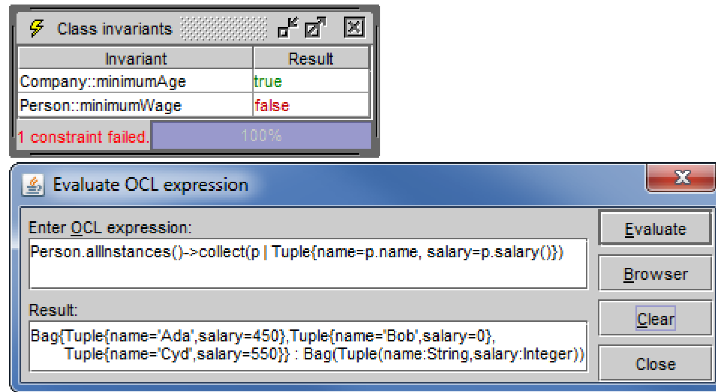

Figure 2: Evaluation of class invariants and a user-defined OCL query expression in USE

of the USE tool is to check for software quality criteria like correct functionality of UML descriptions in an implementation-independent manner. This approach takes advantage of descriptive design level specifications by expressing properties more concisely and in a more abstract way. Such properties can be given by state invariants and operation pre- and postconditions. They are checked by the USE system against the test scenarios, i.e., object diagrams and operation calls given by sequence diagrams, which the developer provides.

USE takes as input a textual description of a model and its OCL constraints. It then checks this description against the grammar of the specification language, which is a superset of OCL, extended with language constructs for defining the structure of the model. Having passed all these checks, the model can be displayed by the GUI provided by the USE system. In particular, USE makes available a project browser which displays all the classes, associations, invariants, and pre- and postconditions of the current model.

The diagrams shown in Figure 1 are provided by USE. The status of the implemented OCL invariants in terms of the given model instance can be examined via a class invariants window (see Figure 2). It reveals the invariant minimumWage to be violated. Since USE allows us to query the current model instance via user-defined OCL expressions, we exploit this feature to further inspect the problem. The result is also shown in Figure 2. Our query calculates a tuple of name and total income for each person. We see that Ada and Bob do not reach the minimum wage of 500. However, since Bob is unemployed, he is disregarded by the invariant.

2.1.4. Domain-Specific Modeling and Languages

Domain-specific modeling (DSM) is an approach for constructing systems that fundamentally relies on employing domain-specific languages (DSLs) to represent the different system aspects in the form of models. A DSL is said to offer higher-level abstractions than a general-purpose modeling language and to be closer to the problem domain than to an implementation-platform domain. A DSL catches domain abstractions as well as domain semantics and supports modelers in order to develop models with a direct use of domain concepts. Domain rules can be incorporated into the DSL in the form of constraints, making the development of invalid or incorrect models much harder. Thus, domain-specific languages play a central role in domain-specific modeling. In order to define a domain-specific modeling language, two central aspects have to be taken into account: the domain concepts including constraining rules (which constitute the abstract syntax of the DSL), and the concrete notation employed to represent these concepts (which can be given in either textual or graphical form). In this paper, we mainly focus on the abstract syntax. The abstract syntax of a domain-specific language is frequently described by a metamodel. A metamodel characterizes the concepts of the domain, the relationships between the concepts, and the restricting rules that constrain the model elements in order to reflect the rules that hold in the domain. Such an approach supports fast and efficient development of DSLs and corresponding tools (for example, translators, editors, or property analyzers).

Let us explain these ideas with an example. We consider a few elements of the well-known relational database language SQL as a domain-specific language and show in the screenshot in Figure 3 how these features would be represented and analyzed with our tool USE. We describe the abstract syntax of the considered SQL elements with a metamodel, which embodies structural requirements in the form of a class diagram together with restricting constraints. We show how this metamodel can be validated and analyzed with usage scenarios.

- An overview of the metamodel for the tiny SQL subset is shown in the project browser in the left upper part of the screenshot and in the class diagram in the lower right part. Classes, associations and invariants are pictured in the browser. From the class diagram we learn that a relational schema (class RelSchema representing an SQL table) has attributes (columns) and that an attribute is typed through a data

Figure 3: USE screenshot of Relational DB Metamodel

type. A relational schema is populated with rows (tuples) in which each attribute gets a value by means of attribute map objects.

- Further rules are stated in the form of invariants which restrict the possible instantiations, i.e., the object diagrams of the metamodel. The names of these invariants are shown in the 'Class invariants' window in the middle of the screenshot. We hide the OCL details but only informally explain the constraint purpose in the order in which the invariants appear: (a) the set of key attributes of each relational schema has to be non-empty, (b) the attributes names have to be unique within the relational schema, (c) each row must have an attribute value for each of its attributes, and (d) each row must have unique key attribute values.

- In the upper part of the screenshot we see a usage scenario in concrete SQL syntax. One table (relational schema) is created, populated by two SQL insert commands and finally modified with an additional SQL update command.

- This usage scenario is represented in the abstract syntax of the metamodel in the form of an evolving object diagram. The screenshot shows only the last object diagram after the SQL update has been executed: (a) after the create command only the four left-most objects (rs1, a1, a2, dt1) are present; (b) after the first insert command the five middle objects (r1, am1, v1, am2, v2) appear, however we will have `v1.content='Ada'`; (c) after the second insert the five right-most objects (r2, am3, v3, am4, v4) will show up; up to this point all four invariants evaluate to 'true'; (d) after the update command the 'content' value of v1 changes (`v1.content='Bob'`) and the evaluation of the invariant keyMapUnique turns to 'false'.

- Let us further explain the impact of the invariants by means of changing the stated object diagram: (a) the first invariant would turn to 'false' if we would say `a1.isKey = false`; (b) the second invariant would turn to 'false' if we would say `a2.name = 'firstName'`; (c) the third invariant would turn to 'false' if we would delete the objects am2 and v2; (d) the fourth invariant would turn to 'true', if we would say `a2.isKey=true`.

- The situation is analyzed with the OCL query shown in the screenshot. The OCL query finds the objects which violate the failing constraints: All objects are returned for which another object with the same key attribute values exists.

Our approach to defining a (domain-specific) RBAC language, which will be explained in the forthcoming parts, follows the principles used above for the tiny SQL subset: Definition of the abstract syntax of the language concepts, and characterization of their dynamic evaluation in the form of a metamodel that consists of a class diagram and restricting constraints.

2.2. Role-Based Access Control and Authorization Constraints

RBAC has been widely used in organizations to simplify access management. Roles are explicitly handled in RBAC security policies. Thereby, security management is simplified and the use of security principles like 'separation of duty' and 'least privilege' is enabled [1]. We now give an overview of (general) hierarchical RBAC according to the RBAC standard [23] which is the fundament of our following RBAC UML approach.

RBAC relies on the following sets: U, R, P, S (users, roles, permissions, and sessions, respectively), $UA \subseteq U \times R$ (user assignment to roles), $PA \subseteq R \times P$ (permission assignment to roles), and $RH \subseteq R \times R$ (partial order called role hierarchy or role dominance relation written as $\leq$). *Users* may activate a subset of the roles they are assigned to in a *session*. P is the set of ordered pairs of *actions* and *resources*. Actions and resources are also called operations and objects in the RBAC context. For disambiguating RBAC and UML concepts, we continuously use the former notion. Resources represent all elements accessible in an information technology (IT) system, e. g., files and database tables. Actions, e. g., 'read', 'write' and 'append', are applied to resources.

The relation PA assigns a subset of P to each role. Therefore, PA determines for each role the action(s) it may execute and the resource(s) to which the action in question is applicable for the given role. Thus, any user having assumed this role can apply an action to a resource if the corresponding ordered pair is an element of the subset assigned to the role by PA.

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.

An important advanced concept of RBAC is authorization constraints. Authorization constraints can be regarded as restrictions on the RBAC functions and relations. For example, a (static) SoD constraint may state that no user may be assigned to both the *cashier* and *cashier supervisor* role, i.e., the UA relation is restricted. It has been argued elsewhere [1] that authorization constraints are the principal motivation behind the introduction of RBAC. They allow a policy designer to express higher-level organizational rules as indicated above. In the literature, several kinds of authorization constraints have been identified. In this paper, we exemplarily consider static and dynamic SoD [3, 2] and cardinality constraints [1]. Temporal considerations need extra preparation which we introduce later.

2.3. Role-based Delegation and Revocation

Delegation is the process whereby an active entity in a distributed environment authorizes another entity to access resources. In current distributed systems, a user often needs to act on another user's behalf with some subset of his/her privileges. Most systems have attempted to resolve such delegation requirements with ad-hoc mechanisms by compromising existing disorganized policies or simply attaching additional components to their applications. There is still a strong need in the large, distributed systems for a mechanism that provides effective privilege delegation and revocation management. To this end, several delegation models have been proposed recently for access control systems [9, 7, 24, 25]. RBDM0 is a model for delegating roles, which is based on the RBAC0 model of the RBAC96 family [9]. RDM2000 defines a rule-based framework for role-based delegation and revocation [7]. The model considers role hierarchies and also provides support for multi-step delegations. The PBDM model proposes a delegation model for permissions that also supports multi-step delegations [24]. In this paper, we address our approach based on well-established RDM2000 delegation and revocation framework.

2.3.1. Role Delegation

In addition to the basic components defined in RBAC model, RDM2000 defines a new relation called delegation relation (DLGT). It includes three elements: original user assignments UAO, delegated user assignment UAD, and constraints. The motivation behind this relation is to address the relationships among different components involved in a delegation. To illustrate main functional components in RDM2000, we use a role hierarchy example

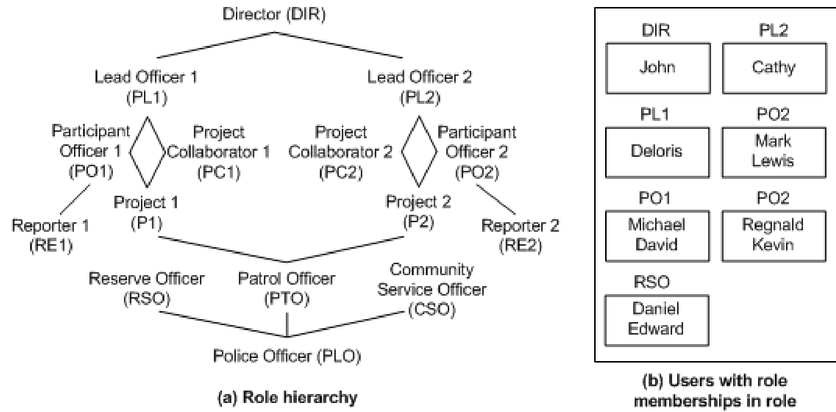

Figure 4: An example of organizational role hierarchy and users.

shown in Figure 4. In a user-to-user delegation, there are four components: a delegating user, a delegating role, a delegated user, and a delegated role. For example, (*Deloris*, PL1, *Cathy*, PL1) means *Deloris* acting in role PL1 delegates role PL1 to *Cathy*. The delegation relation supports role hierarchies: a user who is authorized to delegate a role r can also delegate a role r' that is junior to r. For example, (*Deloris*, PL1, *Lewis*, PC1) means *Deloris* acting in role PL1 delegates a junior role PC1 to *Lewis*. A delegation relation is one-to-many relationship on user assignments. It consists of original user delegation (ODLGT) and delegated user delegation (DDLGT).

In some cases, we may need to define whether or not each delegation can be further delegated and for how many times, or up to the maximum delegation depth. RDM2000 introduces two types of delegation: *single-step delegation* and *multi-step delegation*. Single-step delegation does not allow the delegated role to be further delegated; multi-step delegation allows multiple delegations until it reaches the maximum delegation depth. The maximum delegation depth is a natural number defined to impose restriction on the delegation. Single-step delegation is a special case of multi-step delegation with maximum delegation depth equal to one.

A *delegation path (DP)* is an ordered list of user assignment relations generated through multi-step delegation. A delegation path always starts from an original user assignment.

RDM2000 has the following major components and theses components are formalized from the above discussions.

- $UAO \subseteq U \times R$ is a many-to-many original user to role assignment relation.
- $UAD \subseteq U \times R$ is a many-to-many delegated user to role assignment relation.
- $UA = UAO \cup UAD$.
- $DLGT \subseteq UA \times UA$ is one-to-many delegation relation. A delegation relation can be represented by $((u, r), (u', r')) \in DLGT$, which means the delegating user u with role r delegated role r' to user u'.
- $ODLGT \subseteq UAO \times UAD$ is an original user delegation relation.
- $DDLGT \subseteq UAD \times UAD$ is a delegated user delegation relation.
- $DLGT = ODLGT \cup DDLGT$.

Delegation authorization is to impose restrictions on *which role* can be delegated to *whom*. RDM2000 adopts the notion of prerequisite condition to delegation authorization specification. A prerequisite condition CR is a Boolean expression using the usual "&" (and) and "|" (or) operators on terms of form x and $\overline{x}$ where x is a regular role, for example, $CR = r_1 \& r_2 | \overline{r_3}$.

The following relation authorizes user-to-user delegation in RDM2000:

- $can_delegate \subseteq R \times CR \times N$
  where R, CR, N are sets of roles, prerequisite conditions, and maximum delegation depth, respectively.

The meaning of $(r, cr, n) \in can_delegate$ is that a user who is a member of role r (or a role senior to r) can delegate role r (or a role junior to r) to any user whose current entitlements in roles satisfy the prerequisite condition cr without exceeding the maximum delegation depth n. For example, $(PL1, PO2, 1) \in can_delegate$, then *John* can delegate role *PC1* to *Mark* who is a member of *PO2* role, so that $(John, PL1, Mark, PC1) \in DLGT$

2.3.2. Role Revocation

Revocation is an important process that must accompany the delegation. For example, *Cathy* delegated role *PC1* to *Mark*; however, if *Mark* is transferred to another division of the organization, he should be revoked from the delegated role *PC1* immediately. Several different semantics are possible for user revocation [9, 16]. RDM2000 articulates user revocation in the following dimensions: *dominance*, *propagation*, and *grant-dependency*. *Dominance*

refers to the effect of a revocation; *strong* revocation considers also implicit role memberships (e.g, through role hierarchies), whereas *weak* revocation does not. *Propagation* refers to the extent of a revocation; a *cascading* revocation also revokes all the subsequent delegations. Grant-dependency refers to the legitimacy of a user who can revoke a delegated role. *Grant-dependent (GD)* revocation means only the delegating user can revoke the delegated user from the delegated role. *Grant-independent (GI)* revocation means any original user of the delegating role can revoke the user from the delegated role.

RDM2000 defines the following relations authorizing delegation revocation w.r.t. grant-dependency:

- $can_revokeGD \subseteq R$
- $can_revokeGI \subseteq R$

The meaning of $(b) \in can_revokeGD$ is that only the delegating user who has current membership in b can revoke a delegated user from the delegated role that is junior to b. The meaning of $(b) \in can_revokeGI$ is that any user whose current membership includes a delegated role b in the delegation path that is prior to a delegated user whose current membership includes a delegated role junior or equal to b, can revoke the delegated user from role b. Similarly, revocation relations can be defined for dominance and propagation properties.

3. RBAC UML Description

Three central requirements form the basis of the developed RBAC metamodel. The model must provide for (1) the design of organizational (security) policies with respect to core RBAC concepts including authorization constraints, (2) a comprehensive validation of the specified policies including time-independent (static) and time-dependent (dynamic) aspects, and (3) extensibility.

These requirements result in a UML class diagram with two parts describing a *policy level* for the policy design and a *user access level* for the policy analysis. Figure 5 visualizes the basic idea. An object diagram shows an example instance of the RBAC class diagram. The dark grey part represents a role-based policy specified by an administrator (security officer) through the creation of Role, Permission, Action and Resource objects and insertion

of links between the objects. In this example, no authorization constraints are involved.

In summary, the dark grey part represents the role-based policy as an object diagram, which is based on the RBAC class diagram, i.e., the metamodel. Proceeding this way, all the details of the RBAC metamodel are hidden from a security officer, which leads to a more usable RBAC DSL than a logic-based policy language. This is in line with Jaeger and Tidswell, who pointed out that graphical policy languages were more suitable for an administrator than logic-based languages [26].

The light grey part simulates an IT system with one user bob who is present at two different points in time and his activities. With the help of the object diagram, we can sketch the main principles of RBAC and our UML model, which is examined in detail later. The example policy manages the access to just one resource, a (bank) check. (This is a simplified view to an RBAC permission management. RBAC policies often abstract from individual resources.) Users in the role of a 'clerk' are entitled to prepare checks. Users in the role of a 'supervisor' are allowed to approve them. As mentioned before, policy designers (administrators) normally aim to prevent situations in which the same user prepares and approves a critical resource like a check (SoD requirement).

The user access level is exclusively designed for the analysis of policies including authorization constraints like the aforementioned SoD requirement. The analysis is performed by administrators who can either manually instantiate the user access level or let a UML validation tool (e. g., USE) automatically create user access scenarios. The user access level simulates concrete user activities in the context of a policy, i.e., the actor 'End-user' in Figure 5 represents real users defined by an administrator, but the users' activities are simulations of real events. In the present case, the following situation is at hand. The user bob prepares a check at 10 am and approves this check in a different session at 11 am, thus, violating the SoD requirement. Speaking more precisely, bob accesses the real resource 'check' via the action 'prepare' and later in the context of another access via the action 'approve'. We call a point in time a *snapshot* and a sequence of snapshots a *scenario* or *film strip* to stress the sequential aspect.

The user activity can be checked with respect to the policy. It is either valid, i. e., the whole object diagram fulfills all underlying UML and OCL constraints specified with the RBAC metamodel, or invalid, i. e., the object diagram violates at least one UML or OCL constraint. The UML and OCL

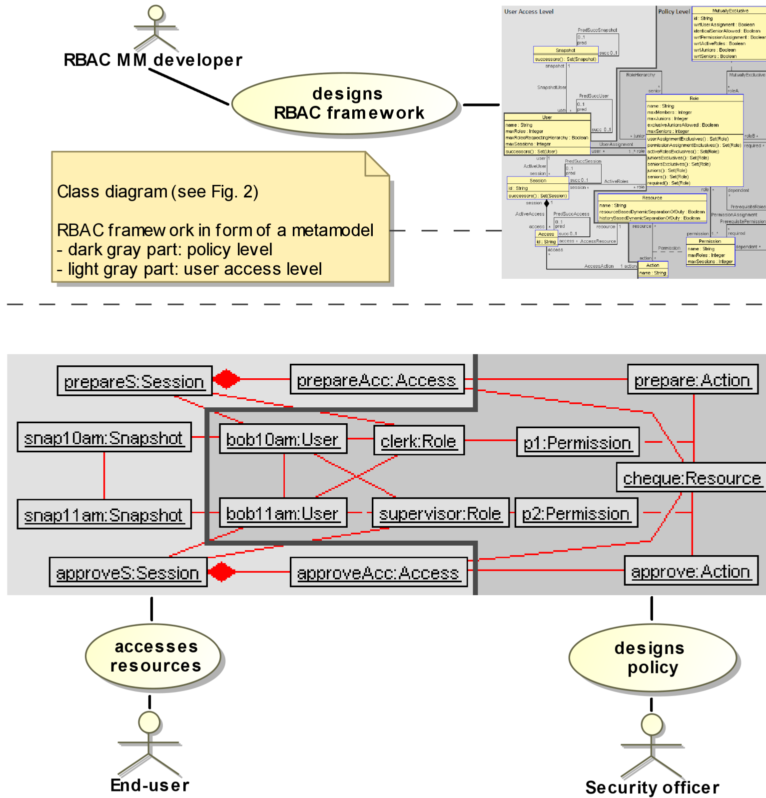

Figure 5: Policy and user access level of the RBAC UML description

constraints are controlled by the policy part as the policy determines the set of active authorization constraints. For example, if the administrator activates the respective SoD authorization constraint (a boolean UML attribute belonging to Resource objects which is currently hidden in the diagram) for the 'check', the OCL invariant enforcing the SoD requirement will come into effect. Thus, the present scenario will not be valid in the context of the restricting policy.

The distinction between the actors, i.e., the RBAC metamodel developers (the authors of this paper), security officers (administrators), and end-users, is helpful later when we address the various possibilities to analyze the RBAC description.

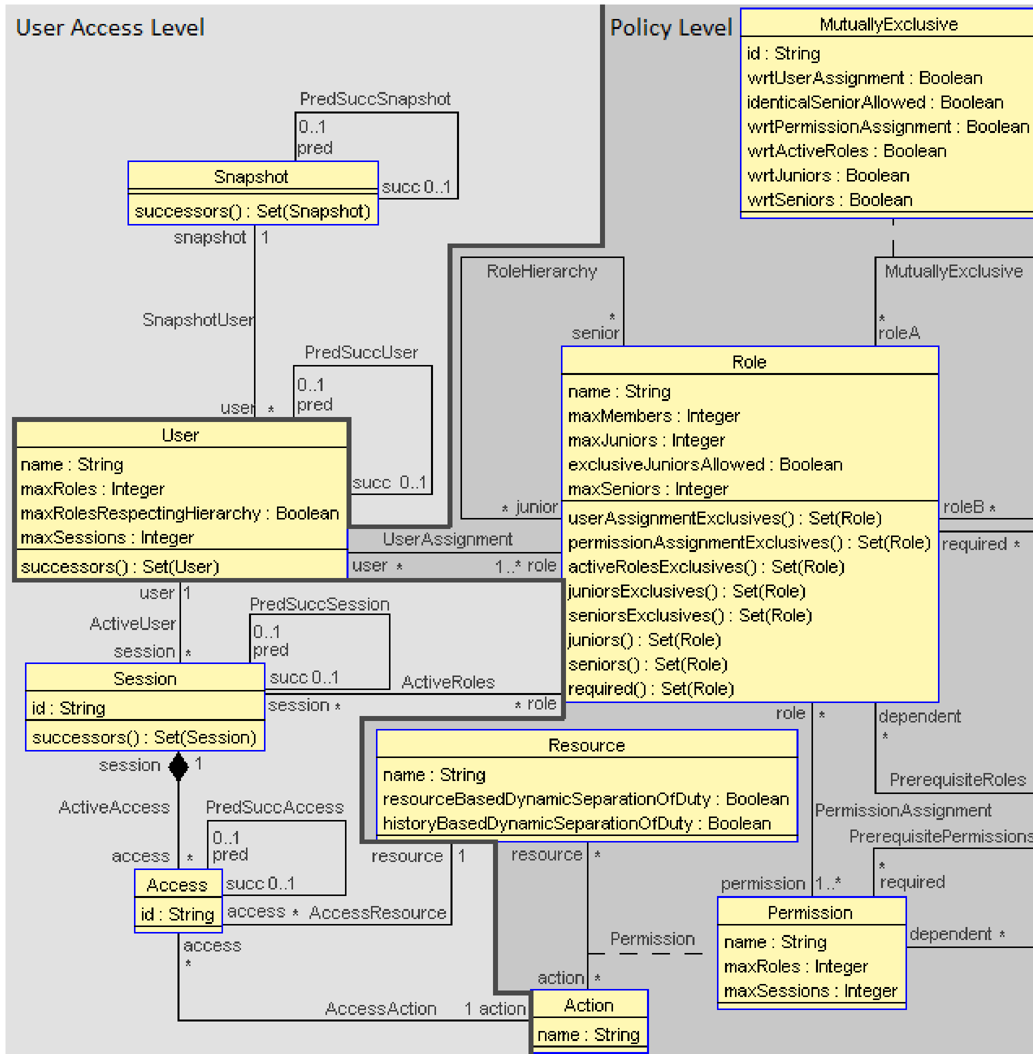

Figure 6: The RBAC metamodel

3.1. RBAC Metamodel

The object diagram shown in Figure 5 is based on the RBAC metamodel shown in Figure 6. Please note that Figure 6 depicts one class diagram, which, however, consists of the two parts policy level and user access level. Classes and associations analogously belong to the policy level or the user access level. This situation is similar to that shown in the bottom right corner of Figure 3, in which a class diagram for the Relational DB metamodel is given. This class diagram describes two conceptually different parts, namely, the definition of the DB schema as well as the modification/selection of DB rows.

In the following sections, we describe both parts of the RBAC metamodel in more detail. In addition, Fig. 7 gives an overview of different use cases, which describe how our RBAC framework can be applied. The MM modeler specifies the UML metamodel of our RBAC DSL including the OCL constraints and the class diagram. Whenever new RBAC concepts are introduced (e.g., new types of authorization constraints or as in this paper, delegation and revocation concepts), the RBAC MM modeler adjusts the metamodel accordingly. Based on the metamodel, object diagrams can be defined, which instantiate the two parts of the RBAC metamodel. A security officer (administrator) defines the role-based policy of her organization and works at the policy level to set attributes (e.g., expressing authorization constraints), add objects (e.g., roles, users, resources), and set links between objects (e.g., defines RBAC relations). The end user then accesses the security-critical resources, such as accounts ("User Access Level Diagram" in Fig. 7). Please note that this level can also be used by a security officer to simulate the system and test the corresponding role-based policy. Also, it can serve as a basis for an implementation of an authorization engine [27].

3.1.1. Policy Level

The dark grey policy part features the basic RBAC concepts. Users are assigned to at least one role. Roles entail a particular set of permissions which are needed for applying actions to resources. The role hierarchy and RBAC authorization constraints form the realized advanced concepts. Roles may have junior roles implying the inheritance of permissions. The authorization constraints are based on the fundamental paper of Sandhu [1] supplemented by dynamic constraints discussed in [28]. In our approach, the constraints are realized as UML attributes and associations.

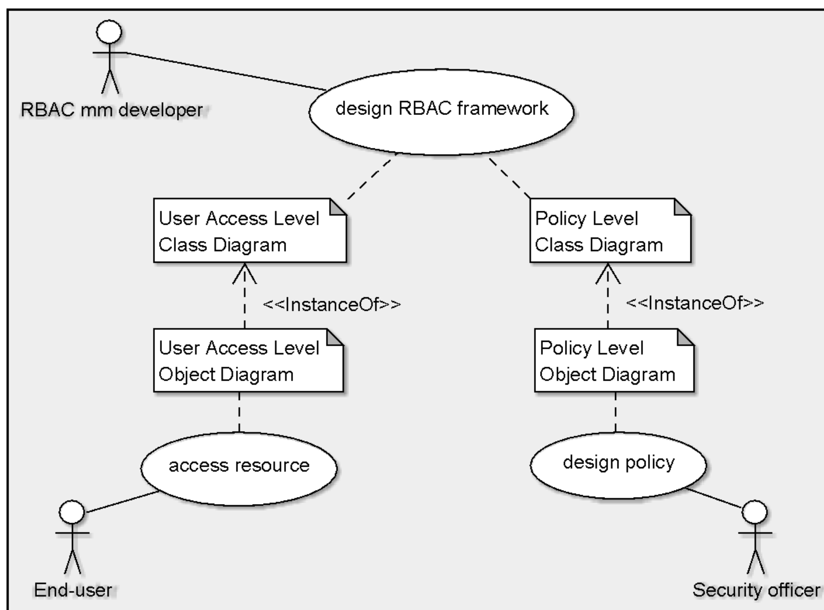

Figure 7: Use cases for policy management with the RBAC DSL

While integrating the authorization constraints into the RBAC metamodel, we adhered to the principle of strictly separating the RBAC metamodel from concrete policies. That is, concrete policies should be exclusively defined in object diagrams so that their specification does not require adjustments at the metamodel level. Generally speaking, our approach allows the policy administrators to freely configure the needed authorization constraints by setting attribute values and inserting links between objects. While the attribute and association names are chosen to suggest the meaning of the corresponding constraint, we provide a short description for each realized authorization constraint within Tab. 1. The OCL invariants implementing the authorization constraints are considered in Sect. 3.2.

3.1.2. User Access Level

As explained before, the user access level displayed in the light grey part of Figure 6 is an essential means for policy analysis. On the one hand, the class 'User' and related authorization constraints belong to the policy level because administrators create users and configure their access rights through the assignment to roles and the determination of the respective attribute values. On the other hand, a user represents a central element at the user

Table 1: Realized authorization constraints

Constraint	Description	Reference
User:: maxRoles	maximum number of roles the user is assigned to (respecting or ignoring the role hierarchy, depending on the boolean value of attribute 'maxRolesRespecting-Hierarchy')	[1], page 11, lines 29–30
maxSessions	maximum number of simultaneously active sessions with respect to a user	[1], page 12, lines 15–16
Role:: maxMembers	maximum number of assigned users	[1], page 11, lines 27–28
maxJuniors	maximum number of inheriting junior roles (mutually exclusive juniors allowed or prohibited, depending on the boolean value of attribute 'exclusiveJuniors-Allowed')	[1], page 12, lines 30–31
maxSeniors	maximum number of senior roles	[1], page 12, lines 30–31
PrerequisiteRoles (Assoc.)	dependent role postulates required role with respect to user assignment	[1], page 11, lines 36–38
MutuallyExclusive::		
wrtUserAssignment	a user must not be assigned to both of the connected roles (identical seniors can be explicitly allowed by setting the boolean attribute 'identicalSeniorAllowed' to true)	[1], page 11, lines 6–7
wrtPermissionAssignment	a permission must not be assigned to both roles	[1], page 11, lines 10–12
wrtActiveRoles	the connected roles must not be both activated in a session (possibly involving several snapshots)	[1], page 12, lines 14–15
wrtJuniors	the connected roles must not have the same junior roles	[1], page 12, lines 31–32
wrtSeniors	the connected roles must not have the same senior roles	[1], page 12, lines 31–32
Permission:: maxRoles	maximum number of roles the permission is assigned to	[1], page 11, lines 30–32
maxSessions	maximum number of sessions simultaneously activating the permission (i. e., within the same snapshot)	[1], page 12, lines 16–17
PrerequisitePermissions (Assoc.)	assignment of the dependent permission postulates the assignment of the required permission	[1], page 12, lines 1–3
Resource::		
resourceBasedDynamic-SeparationOfDuty	a user may not apply more than one action to the resource	[2], page 4, line 16–20
historyBasedDynamic-SeparationOfDuty	a user may not apply all available actions to the resource	[2], page 4, line 28–39

access level because we model the users' activities via sessions and resource accesses at this level. In other words, a User object is part of a concrete policy, but the activated sessions and accesses related to the User object simulate an IT system which underlies the designed policy. This way, during the analysis process, we can, for example, identify user activities which are forbidden by the given policy specification, but are valid in the eyes of the administrators, or identify constellations which are allowed wrt. the policy but should actually be forbidden.

The policy level of the RBAC UML description follows the principles of an *application model*, whereas the user access level follows the principles of a *snapshot model* [29, 30]. That is, one object diagram for Figure 6 describes exactly one policy, but several situations on the user access level, i.e., points in time in a IT system. The class 'Snapshot' and the associations with 'Pred-Succ' prefix enable the corresponding dynamics. A scenario consists of one chain of successive snapshots. Analogously, users, sessions and accesses can have successors. These predecessor/successor relationships allow for identifying the individual users, sessions and accesses over time (snapshots). For example, the user Bob is represented by one object per snapshot so that we can follow Bob's activities within the whole scenario. This aspect is not explicitly treated in [30].

This snapshot modeling of the user access level with pred/succ associations allows us to analyze time-dependent (dynamic) constraints.

3.2. Supplemental OCL Constraints

The RBAC class diagram is supplemented by OCL invariants which serve three purposes. They (1) represent authorization constraints, (2) check for reasonable policy designs, and (3) regulate the snapshot concepts.

The OCL invariants make use of OCL query operations displayed in the operation parts of the classes (see Figure 6). The query operations represent auxiliary functions simplifying the invariant bodies or calculating transitive closures. For example, the operation 'successors' (Snapshot) returns all direct and indirect successors of the snapshot under consideration, or the operation 'required' (Role) calculates all directly and indirectly required roles in the context of the calling Role object.

3.2.1. Formalizing Authorization Constraints

Each authorization constraint is represented by an OCL invariant which checks whether a user access scenario complies with the authorization con-

straint. The administrator determines for which objects the authorization constraint should be activated, i. e., for which objects the invariant should be applied. This is done by creating objects on the policy level, changing attributes, or establishing links. The invariant corresponding to the authorization constraint comes into play through these modifications. Please note that the invariant is formulated only once, and can be activated in different contexts. For example, consider the invariant 'MaximumNumberOfMembers' stated below. It corresponds to the authorization constraint which is configured with the attribute 'maxMembers' of class 'Role'. After determining a value for 'maxMembers' in the context of a Role object in the policy, the related invariant is activated which checks the requirement for the Role object.

```
context r:Role inv MaximumNumberOfMembers:
  r.maxMembers.isDefined implies r.user->size() <= r.maxMembers
```

This invariant expresses a static, time-independent property because it must hold at each point in time. In contrast, the invariant 'NoExclusiveRolesActive' related to the (switch) attribute 'wrtActiveRoles' of class 'MutuallyExclusive' has to respect the snapshot framework.

It ensures that no pair of roles exists which are characterized as mutually exclusive with respect to the activation in a single session. That is, the attribute 'wrtActiveRoles' is set to 'true', and it is used in the definition of the query operation 'activeRolesExclusives,' which is used in the following invariant:

```
context s:Session inv NoExclusiveRolesActive:
  let activeRoles = s.successors().role->union(s.role) in
  activeRoles->excludesAll(activeRoles.activeRolesExclusives())
```

As sessions are active in an arbitrary time frame, they often persist several snapshots until the respective user terminates them. Hence, the invariant must include the whole time frame wrt. a session, i.e., the sequence of successive Session objects (s.successors()), representing the single considered session over time.

3.2.2. Checking for Reasonable Policies

The model comprises further invariants assisting the administrators (at a syntactical level) to design correct policies. Thus, structurally inconsistent

Table 2: Different perspectives of analyzing RBAC

RBAC level	**Focus**	**Analyzed by**	**Considered subject**
RBAC metamodel	class diagram and OCL constraints	RBAC DSL developers	all instantiable policies all possible RBAC **scenarios**
RBAC policy	**static** policy aspects	policy administrators	one specific (partial) policy all possible RBAC **snapshots**
	dynamic policy aspects	policy administrators	one specific (partial) policy all possible RBAC **scenarios**
User access	resource access	authorization system (based on an RBAC policy)	one specific RBAC **scenario**

policies, e. g., showing self excluding roles or roles which simultaneously require and exclude themselves, can be avoided in the first place. The aim is to allow the administrators to focus on semantical aspects, like assigning the end-users to proper roles so that they achieve a policy which matches their intended security properties.

3.2.3. Constraining User Access Scenarios

Finally, a set of OCL invariants is created to maintain valid sequences of snapshots. For example, only one scenario is allowed within an object diagram and the set of snapshots must be properly ordered. All sources related to the RBAC metamodel can be found in [31].

4. Analyzing the RBAC Description

If we consider the complexity of real RBAC policies and the extensive possibilities of designing a policy by means of the RBAC metamodel, and if we consider the resulting possibilities of overlooking security holes, we see that computer-aided analysis is essential at the policy level. In the following, we discuss several ways to validate the RBAC description given above. This discussion is based on our earlier work [18].

As an adequate RBAC metamodel is the precondition for designing accurate policies, the model itself must be sound. Regarding the number of classes, associations and attributes as well as the number of OCL constraints, the RBAC UML model has reached a size which makes pure manual validation impossible. Thus, the UML and OCL experts who maintain the RBAC metamodel (the DSL) within an organization (as well as the authors of this paper) also need tool support. Table 2 shows the different approaches to analyzing the RBAC artifacts including the RBAC metamodel, RBAC policies and the user access. In the following, the user access level can be disregarded

because user activities are restricted by a policy. Consequently, a complete and correct policy suffices to enable only valid user activities.

4.1. The USE Model Validator

Our RBAC description provides diverse interfaces for analysis so that any UML and OCL tool with analysis functionality can help to ensure a sound RBAC metamodel and well-designed policies. We follow the approach of the USE system [15]. In order to ensure properties of the metamodel or the policies, we search system state spaces, i. e., sets of object diagrams. The existence of an object diagram fulfilling specified conditions gives information about the model or the policy characteristics.

The success of this approach strongly depends on the performance of the underlying search engine. In [28], we employ the ASSL generator [15] integrated into USE to analyze RBAC policies in order to detect missing and conflicting static authorization constraints. The enumerative generator has to consider all possible object diagrams in the worst case, i.e., if there is no state having the required properties. Hence, it cannot handle models of the size of the present RBAC metamodel with acceptable execution times. The developed USE model validator resolves this problem. It is based on the relational model finder Kodkod representing the successor of the Alloy Analyzer [32]. Both tools provide a relational logic for specifying and analyzing models. Internally, they translate the model and properties to be checked into a SAT problem which can be handled by any SAT solver. Kodkod is designed as a Java API simplifying the integration into other tools.

The model validator includes a translation from UML and OCL concepts into relational logic. The current version comprises all important UML class diagram and OCL features. As the RBAC metamodel is completely supported, it can be taken as an example for the successful use of the model validator, see [31] for details.

4.2. Analyzing the RBAC Metamodel

A comprehensive analysis of the RBAC metamodel during development helped us to discover several unwanted properties. Also future extensions of the model with respect to further RBAC features will benefit from further analysis of the model properties. Our examinations presented here are based on the core concepts of *independence* and *reasoning* discussed in [33].

4.2.1. Independence

The independence of constraints describes the fact that each defined constraint adds essential information to the model, i. e., it further restricts the space of valid object diagrams. This property can be checked by searching for an object diagram fulfilling all constraints but the constraint under consideration. If such a diagram exists, the respective constraint is independent from the others because it does not follow from them. Each check results in an object diagram or yields no solution. The latter case indicates dependencies between the constraints which have to be further examined, e. g., by temporarily disabling not involved constraints. Within the RBAC metamodel given above, all constraints are independent. The results for each invariant are presented in [31].

4.2.2. Reasoning

Reasoning stands for the universal examination of model properties. Properties under consideration are often complex, but in many cases simple properties already lead to the desired information. For example, in order to check a specific RBAC metamodel invariant, we can configure the model validator to search for a valid object diagram, in which the authorization constraint corresponding to the invariant is activated. Proceeding this way, we discovered a further erroneous invariant during development. We defined a search space, from which we expected to find object diagrams which satisfy the invariant, but finally found none. More details on this situation can be found in the paper from Kuhlmann et al. [18]

4.3. Analyzing RBAC policies

Complex security policies usually become opaque with respect to their implicit properties, i. e., the combination of the explicitly stated authorization constraints often yields new properties which have to be analyzed. Consequently, changes to a policy may have various effects. Even simple policies like the ones presented in this section can reveal unanticipated characteristics. In the context of our RBAC metamodel and the model validator these characteristics can be uncovered by searching for specific object diagrams. In contrast to the analysis at the metamodel level, the analysis of policies is normally based on a given object diagram representing the policy under consideration or a partial policy which may be automatically adapted during the search. That is, administrators can determine which parts of the designed policy should be fixed (e. g., permission 'p1' must be assigned to role

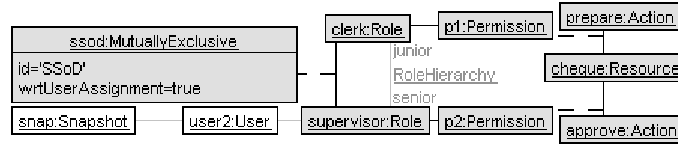

Figure 8: Partial policy and partial search results (white objects, grey links)

'clerk' and the number of roles must not change) or are variable (e. g., the user assignment to roles can arbitrarily be changed during the search). In many cases, at least some parts of a policy remain variable.

The analysis with the model validator needs two artifacts, an object diagram—the (partial) policy—and a property to be checked. The property can be formulated in the form of a usually non-complex OCL expression and by explicitly stating the bounds with respect to the number of objects and links for each class and association as well as the definition of attribute values. Let us take the object diagram shown in Figure 8 which presents the first artifact, a partial policy (grey objects and black links) with some fixed elements, e. g., the existing objects must not be deleted, users do not change their roles, and the attribute value of 'wrtUserAssignment' must remain 'true', i.e., a user may not have both roles 'clerk' and 'supervisor' at the same time. The white objects and grey links are not part of the policy. They are addressed later. Please note that we manually adapted the displayed object diagram to combine the elements existing before and after the search. The second artifact represents the following property to be checked (informally): 'Does the policy allow a user to apply both actions ('prepare' and 'approve') to the resource in the context of a snapshot, although a user cannot have both roles?' Modeling this property with OCL, we require (among other requirements) the following statement to be fulfilled.

```
User.allInstances()->exists(u |
  u.session.access.action->asSet()->size() = 2)
```

These kinds of statements normally have specific patterns which are often reused in case of other properties. Thus, the administrators do not need to have a deep insight into the OCL semantics. Moreover, the patterns could be enforced and implemented in the used UML tool (e.g., USE) in order to allow property configurations through a graphical user interface.

Giving both artifacts to the model validator, it returns a completed object diagram fulfilling all constraints. It is partly shown with the white objects

and grey associations in Figure 8. We hide the overhead like the session in which the user accesses the resource via both actions. We see that the static SoD property is circumvented, if the role 'clerk' becomes the junior role of 'supervisor' because a supervisor will in turn inherit all permissions from a clerk.

Beside the automatically generated object diagrams, it is also often helpful to manually specify scenarios of user activities. They can, for example, be used as positive (valid object diagrams) and negative (invalid object diagrams) test cases during the development of policies. When a reasonable set of test cases is available, it can be periodically checked during the development process because a failed test can indicate the existence of a new unwanted property within the policy, possibly resulting from the interplay of several authorization constraints. However, if a policy undergoes great structural changes, the test cases must be adapted accordingly. We give further examples of such test cases in the context of the discussion of the banking application (see Section 6).

5. UML Description for RDM2000

In the following, we describe how we can extend the UML-based metamodel for RBAC to represent the RDM2000 features, which have been presented in Section 2.3. The RBAC metamodel depicted in Figure 5 serves as a basis as RDM2000 is based upon RBAC96. The class diagram shown in Figure 9 represents the extension of the UML metamodel for RDM2000. To better understand how our metamodel captures the RDM2000 concepts, Table 3 gives an overview of the mapping between the RDM2000 features and the corresponding elements of the metamodel.

We have introduced the class `Delegation`, which is used for expressing the ODLGT and DDLGT relations. For example, the delegating pairs of users and roles are expressed by the associations `DelegatingUser` and `DelegatingRole`, respectively; delegated users and roles can be represented similarly by the `DelegatedUser` and `DelegatedRole` associations. The instances of the reflexive association `delegatedDelegation` indicate that we have a DDLGT relationship; if, however, a `delegatedDelegation` link is absent, we have an ODLGT relationship.

RDM2000 lets one impose restrictions on delegation authorization through the *can_delegate* relation (see Section 2.3.1). The maximum delegation depth, for example, is represented by the `maxDelegationDepth` attribute of the `Role`

Figure 9: The delegation metamodel.

class. In addition, we need an OCL invariant, expressing the fact the delegation depth must not be exceeded (also see Table 3). For other modeling elements of our metamodel, additional OCL invariants are also required. We only give the invariant for the `maxDelegationDepth` attribute here as an example: `context d:Delegation`

```
  inv MaxDelegationDepth:
    let maxDepth = d.getDelegationPath()->first()
      .delegatingRole.maxDelegationDepth in
    (maxDepth.isDefined() implies
      d.getDelegationPath()->size()-1 <= maxDepth)
    and
    (d.maxDelegationDepth.isDefined() implies
      d.getNumberOfFurtherDelegations() <= d.maxDelegationDepth)
```

This constraint comprises two parts, which must be checked. First, it must be made sure that the current delegation `d` does not violate the maximum delegation depth of the (original) delegation starting the delegation

Table 3: Mapping between RDM2000 features and the corresponding metamodel elements.

RDM2000 concept	**metamodel element**
UAO	`OriginalUserAssignment` association
UAD	`Delegation` class with the associations `DelegatedUser` and `DelegatedRole`
ODLGT	`Delegation` class with the associations `DelegatingUser`, `DelegatingRole`, `DelegatedUser`, and `DelegatedRole`, invariant `DelegatingUserOrDelegation`
DDLGT	`Delegation` class with the associations `DelegatingUser`, `DelegatingRole`, and `DelegatedDelegation`, invariant `DelegatingUserOrDelegation`
Delegation authorization: maximum delegation depth	`maxDelegationDepth` attribute of the classes `Role` and `Delegation`, `MaxDelegationDepth` invariant
Delegation authorization: delegating role	class `DelegationAuthorization`, association `AuthorizationRole`, invariants `DelegationAuthorizationWrtOriginalRole` and `DelegationAuthorizationWrtDelegatedRole`
Delegation authorization: prerequisite condition	class `DelegationAuthorization`, association `DelegationPrerequisiteRoles`, invariant `DelegationAuthorizedWrtConditions`
Delegation authorization: negation	class `DelegationAuthorization`, association `DelegationForbiddenRoles`, invariant `DelegationAuthorizedWrtConditions`
Delegation authorization: `&`-operator	class `DelegationAuthorization`, association `DelegationPrerequisiteRoles`, invariant `DelegationAuthorizedWrtConditions`. Explanation: Each prerequisite role link represents a conjunct of the form 'the delegated user must have the linked role' and each forbidden role link represents a conjunct of the form 'the delegated user must not have the linked role'.
Delegation authorization: `|`-operator	class `DelegationAuthorization`, association `DelegationPrerequisiteRoles`, invariant `DelegationAuthorizedWrtConditions`. Explanation: Each `DelegationAuthorization` object linked to a specific role represents a disjunct.
Grant-dependency	attribute `grantDependentRevocation` of class `Role` and invariant `GrantDependency`
Dominance	attribute `strongRevocation` of class `Role`, invariant `ActionsPermitted` with the query operation `isRevoked()`
Propagation	attribute `cacadingRevocation` of class `Role`, invariant `ActionsPermitted` with the query operation `isRevoked()`

path. Second, if the current delegation has also set a maximum delegation depth, it must be guaranteed that the subsequent delegations of `d` also respect `d`'s maximum delegation depth.

The prerequisite roles, which the delegated user must possess on delegation, are expressed in our approach with the help of the `DelegationAuthorization` class and the `DelegationPrerequisiteRoles` association. Roles which the delegated user must not possess on delegation are represented by the `DelegationForbiddenRoles` association. All `DelegationForbiddenRoles` and `DelegationPrerequisiteRoles` links connected to a `DelegationAuthorization` object express a conjunct of prerequisite and forbidden roles, which realizes the & operator of RDM2000 (also see Table 3).

The `AuthorizationRole` association also allows one to express alternative prerequisite restrictions on delegation (realizing the "|" operator in the RDM2000 model), i.e., only one of the prerequisite conditions must be satisfied for a successful delegation. Alternative prerequisite conditions can be expressed by multiple instances of the `DelegationAuthorization` class, which are connected to the authorizing role of a delegation via links of type `AuthorizationRole`. Summarizing, we obtain a disjunctive normal form for prerequisite conditions. The conjuncts are represented by all the links of the types `DelegationPrerequisiteRoles` and `DelegationForbiddenRoles` connected to an object of type `DelegationAuthorization` (see the preceding paragraph); the disjuncts are expressed by all connections between a single role (the authorizing role of a delegation) and such `DelegationAuthorization` objects.

In order to realize the semantics of the aforementioned prerequisite conditions for delegation (i.e., the disjunctive normal form), we also need OCL constraints, which are listed in Table 3. For the sake of brevity, we do not present them in this paper. The interested reader is referred to our OCL specification [34], which can be downloaded and then invoked by the freely available USE system.

The type of the revocation scheme (grant-dependency, dominance, propagation) is represented by the respective attributes of the class `Role`. As a consequence, the *type of revocation* (e.g., with respect to dominance, propagation, and grant-dependency) cannot be selected by the revoking user, but only by the policy designer/administrator according to our model. We feel that otherwise the burden of security decisions would be placed on end users (such as physicians) who often do not understand in detail the security implications of their actions. Revocation is expressed by the association `Revocation`, with the associated user being the revoking user. Please note that a revocation does not mean in our model that the delegation links are removed; only a revocation link (i.e., an instance of the `Revocation` association) is added between the revoked delegation and the revoking user. Proceeding this way, the whole delegation history is preserved and can be reconstructed despite revocation actions.

Figure 9 represents the user access level as well as the policy level. The latter is presented in dark grey, whereas the former is in light grey. The user access level describes the user activities, i.e., performing delegation and revocation steps. In contrast, the policy level encompasses all administrative activities, such as constraining delegation through prerequisite rules or determining

the kind of revocation (i.e., setting the attribute values `strongRevocation` or `cascadingRevocation`).

An OCL operation which is central for the specification of revocation is the `isRevoked()` operation defined in the context of the `Delegation` class. We give the OCL specification of `isRevoked()` in the following:

```
context d:  Delegation
isRevoked(curSnap:Snapshot):  Boolean =
  revokingUser.isDefined and
  self.revokingUser.snapshot.successors()->
      including(self.revokingUser.snapshot)->includes(curSnap) or
  delegatedUser.getAllDelegations()->select(d |
    d.getDelegationPath()->first().delegatingRole.strongRevocation and
    d.delegatedRole.seniors()->includes(self.delegatedRole))
                                    ->exists(d|d.isRevoked(curSnap)) or
  getDelegationPath()->first().delegatingRole.cascadingRevocation and
   getDelegationPath()->excluding(self)->exists(d|d.isRevoked(curSnap))
end
```

The operation `isRevoked()` evaluates to true in three cases, which are discussed in the following:

1. The delegation `self` is revoked in the current snapshot `curSnap` or has been revoked earlier. To put this in another way, the current snapshot `curSnap` lies in the present/future of a point in time in which the revocation has been carried out.
2. Considering the delegation `self`, the delegated user's delegated role or a junior role has been revoked strongly (recursive call of `isRevoked()`). This means that the delegated role is also revoked if it is senior to a strongly revoked role for a delegated user.
3. If the cascading revocation has been activated earlier on the delegation path of `self`, the delegated role has been revoked (recursive call of `isRevoked()`).

Another central invariant is `ActionsPermitted`, which makes sure that a user can only execute permitted actions within a session, i.e., actions which a user has obtained through her current roles:

```
context Session:s
 inv ActionsPermitted:
```

```
s.access->forAll(a |
 let
  neededPermissions = a.action.permission->
                                  select(p|p.resource=a.resource)
 in
  a.action.resource->includes(a.resource) and
  s.role.permission->asSet()->includesAll(neededPermissions))
```

This invariant checks (1) if the action can be performed on the resource at all (`a.action.resource->includes(a.resource)`) and (2) if the permissions obtained through the roles activated in this session are sufficient to execute the action. Then, we need an additional constraint which makes sure that a user can only access permissions in a session, which belong to roles that the user holds:

```
inv ActiveRolesSubsetUserRoles:
 s.user.getAllRoles()->includesAll(s.role)
```

The operation `getAllRoles()` collects all the roles which a user possesses, i.e., directly assigned roles, junior roles, and in particular, delegated roles. Due to the fact that `getAllRoles()` must respect revocation, the aforementioned `isRevoked()` operation is also called.

To make our concepts clearer, Figure 10 shows an object diagram, which is based on the metamodel and serves the purpose of explaining our modeling approach. It presents a concrete delegation and revocation scenario of a system including the time steps, in which delegation and revocation activities have been carried out.

In particular, we consider the following situation:

1. `Ada` delegates at point in time `snap1` role `r1` to user `Bob`.
2. `Bob` further delegates in point in time `snap2` role `r1` to `Cyd`.
3. Ada then revokes `r1` from Bob in point in time `snap3`.

We also define the delegation rule: "Role `r1` may be delegated (1) if the delegated user possesses role `r2`, but not role `r3`, or (2) if she possesses role `r4`, but not role `r5`"

Taking a closer look at Figure 10, we can see the aforementioned three steps, which are represented by the snapshots `snap1`, `snap2`, and `snap3`. `d1`

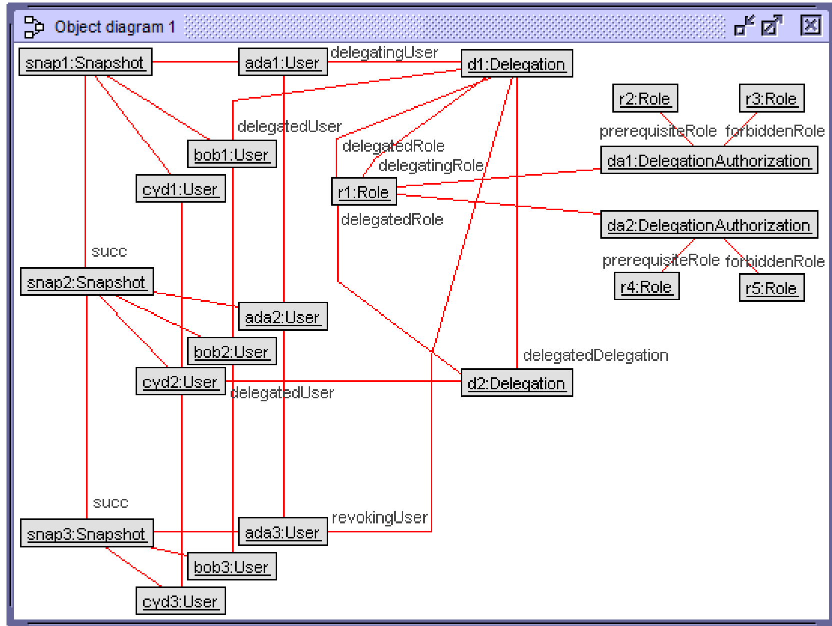

Figure 10: A concrete delegation and revocation scenario represented by an object diagram.

represents the first delegation step (delegation from `Ada` to `Bob`), which is an original delegation. The delegation object `d2` represents the second delegation, which is a delegated delegation. The revocation step is then carried out within `snap3`; here, a corresponding link between the delegation `d1` and the revoking user `Ada` is added to the object diagram. The delegation constraints (delegation policy) are displayed at the right-hand side of the object diagram, expressing the aforementioned delegation rule. Please note that we use two objects of type `DelegationAuthorization` to express the prerequisite rule by simulating the "|" operator. Considering the formalism of the RDM2000 model, this corresponds to the prerequisite condition $r1\&\overline{r3} \mid r4\&\overline{r5}$, which is in disjunctive normal form.

6. Evaluation of our Approach with a Banking Application

In this section, we illustrate the concepts discussed in the previous sections with a banking application, which has been derived from from [17]. In particular, we present several validation scenarios employing the USE tool

as well as the USE model validator, and we show how to use our UML-based DSL to formulate role-based delegation and revocation policies.

6.1. An Overview

The banking application can be used by various bank officers to perform transactions on customer deposit accounts and customer loan accounts, and to generate and to verify financial account data. The roles in the banking system contain *teller*, *customer service representative*, *loan officer*, *accountant*, *accounting manager*, *internal auditor* and *branch manager*. The permissions assigned to these roles include (a) create, delete, input, or modify customer deposit accounts, (b) create, or modify customer loan accounts, (c) create general ledger report, and (d) modify, or verify the ledger posting rules.

The participating roles and permissions performed by each role in the banking system are defined as follows:

1. *teller* - input and modify customer deposit accounts.
2. *customerServiceRep* - create and delete customer deposit accounts.
3. *loanOfficer* - create and modify loan accounts.
4. *accountant* - create general ledger reports.
5. *accountingManager* - in addition to the inherited privileges from *accountant*, modify ledger posting rules.
6. *internalAuditor* - verify ledger posting rules.
7. *branchManager* - perform all privileges of other roles under the emergency case.

Since some roles may perform the privileges of others, there exist dependencies between roles. These dependency relations can be expressed by the role hierarchy. Figure 11 shows the role hierarchy structure in the banking application. The *accountingManager* role is senior to the *accountant* role and the *branchManager* role is senior to all other roles.

In the banking application, several organizational authorization rules should be enforced to support common security principles such as separation of duty and least privilege. We address these authorization rules in the banking application as follows:

- **Rule 1** Some bank officers, such as *teller* and *accountant*, cannot be performed by the same user (see below for the concrete SoD constraints).

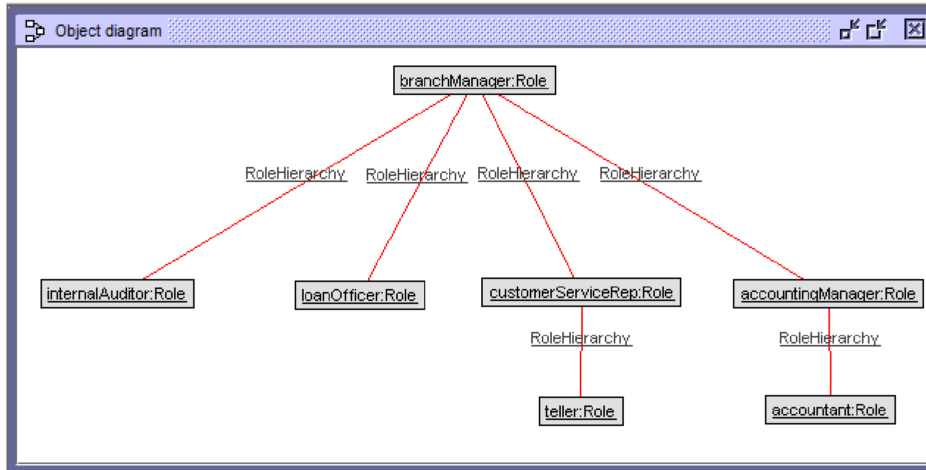

Figure 11: Role hierarchy for banking authorization system.

- **Rule 2** Some users cannot act as the same bank officer.
- **Rule 3** Some bank officers, such as *customerServiceRep* and *loanOfficer*, cannot be activated by the users in the same transaction session.
- **Rule 4** A user can play the bank officer role only if the user has been assigned to another specific bank officer role. For instance, *teller* is a prerequisite role of *customerServiceRep*.
- **Rule 5** The number of users assigned to the bank officer role should be restricted. For example, only *one* user can be assigned to the *internalAuditor* role.
- **Rule 6** When a user is on a business trip or long-term absence, the user can temporarily delegate her authority to others.
- **Rule 7** A user can revoke the delegated authority from others.

6.2. The RBAC policy for the banking application

In the following, we describe several authorization constraints in more detail, which are a part of banking system's RBAC policy.

Authorization constraints. We have discussed above several examples of organizational authorization rules for the banking application. These organizational authorization rules are represented and enforced in RBAC systems

by means of authorization constraints. RBAC constraints including static separation of duty (SSD), dynamic separation of duty (DSD) [2], prerequisite conditions, and cardinality rules can be used to support these organizational authorization rules. Table 4 shows six typical RBAC constraints and corresponding organizational authorization rules for the banking application.

Table 4: RBAC constraints and corresponding rules

Constraint		Supported rule
Statical SoD constraint	SSDRole	`Rule 1`
	SSD User	`Rule 2`
Dynamical SoD constraint	DSD	`Rule 3`
Prerequisite constraint	Prerequisite-Role	`Rule 4`
Cardinality constraint	Cardinality-Role	`Rule 5`

SSD-Role constraints: For the banking system, the following pairs of roles are conflicting:

```
{(customerServiceRep, accountingManager),
(customerServiceRep, internalAuditor),
(loanOfficer, accountingManager), (loanOfficer, internalAuditor),
(accountingManager, internalAuditor), (teller, accountant),
(teller, loanOfficer), (teller, internalAuditor),
(accountant, loanOfficer), (accountant, internalAuditor)}
```

DSD constraints: For the banking system, the following pair of roles is in DSD relation:

`{(customerServiceRep, loanOfficer)}` .

Prerequisite-Role Constraints: For the banking system, the following pair of roles is in a prerequisite role constraint:

- The `Teller` role is a prerequisite role for the `customerServiceRep` role.

Cardinality-Role Constraints: A cardinality constraint can be defined as follows:

- The maximum number of users that can be assigned to `branchManager` and `internalAuditor` is '1'.

[2] As common in the literature on RBAC, we understand DSD in the sense of mutually exclusive roles which must not be activated by a user [23].

Similarly to our earlier works [12, 18], we now formulate the RBAC policy for the banking application in our RBAC DSL, i.e., as a UML object diagram (see Figure 12). Specifically, one can see the roles and the permission assignments as well as role hierarchy relations (e.g., between the roles `accountant` and `accountingManager`). An administrator can also formulate authorization constraints by setting the corresponding attribute values for instances of classes and association classes, respectively. For example, the SSD constraint between the roles `teller` and `accountant` is represented by the association class instance `t_A_SSD_exclusive`. Since it is a static SoD constraint w.r.t. user assignment, the attribute value `wrtUserAssignment` is set to `true`. The other SSD constraints mentioned above are defined similarly with our graphical RBAC DSL and are not shown for the sake of clarity. This is one advantage of our DSL approach; we can hide parts of the policy, such as authorization constraints and permission assignments, to concentrate on those aspects of the policy which are currently interesting for an administrator.

Delegation authorization. In some cases, users in the banking system need to grant their authority to others when they are on a business trip or long-term absence, or need to collaborate with others. Such cases need temporary delegation of authority, for example,

```
can_delegate(customerServiceRep, teller, 1).
```

This delegation rule means that a user who is a member of role *customerServiceRep* (or a role senior to *customerServiceRep*) can delegate the role *customerServiceRep* to a user who is a member of the *teller* role without exceeding the maximum delegation depth of '1'.

Revocation authorization. Revocation is an important process that accompanies the delegation, for example,

```
can_revokeGD(customerServiceRep).
```

The meaning of this revocation constraint is that only the delegating user who has current membership in *customerServiceRep* can revoke a delegated user from the delegated role *customerServiceRep*. Moreover, the revocation is grant-dependent, i.e., only the user who has delegated the role can revoke it.

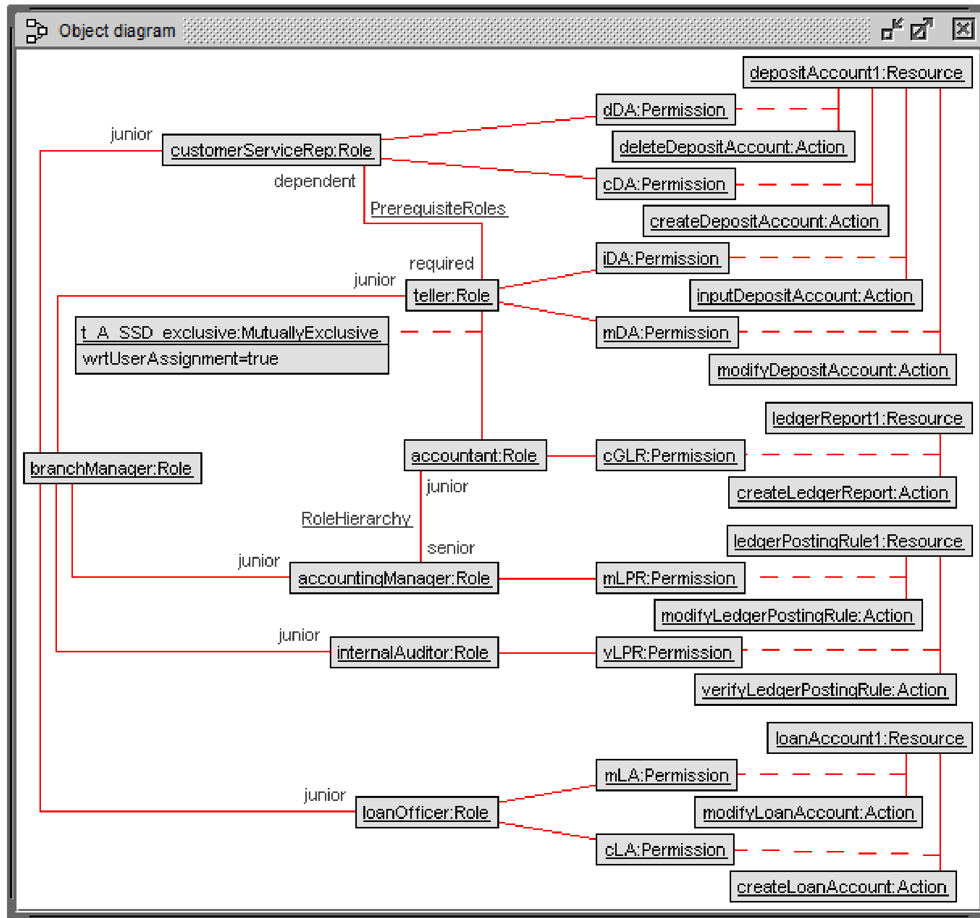

Figure 12: The RBAC policy of the banking application formulated in the UML-based DSL.

6.3. RBAC Policy Analysis

Bringing in authorization constraints, role hierarchies as well as delegation and revocation in access control systems gives rise to the problem of possible conflicts. In other words, when the objectives of two role-based authorization constraints cannot be fulfilled simultaneously, the enforcement of one constraint causes the violation of another constraint. Therefore, it is critical for authorization management to ensure that an authorization constraint is not in conflict with other existing constraints through policy analysis. In this section, we demonstrate how our approach and tools, specifically, USE and the USE model validator, can be leveraged to validate role-based poli-

cies in order to discover subtle policy conflicts. An administrator can employ USE to define positive and negative test cases for a policy as UML object diagrams. Positive test cases describe situations (system states), which the policy should allow, whereas negative test cases should be forbidden by the policy, such as unallowed access to resources. The USE system then confirms or refutes the expectations of the administrator.

In addition, an administrator can define properties for the policy, formulated in OCL. Then the USE model validator can be employed to automatically generate system states that satisfy both the policy and the properties. If the property is not desirable and the USE model validator finds a solution, then the policy is incorrect w.r.t. the expectations of the administrator. If she, however, expects a solution and the validator does not find any, then this lets one conclude that we have a too strict policy.

In the following, we describe three delegation and revocation scenarios to illustrate our validation approach. We also show in this context how to use our DSL to define delegation and revocation policies.

6.3.1. Scenario 1: Conflict between Delegation and SoD Constraints

A delegation constraint may conflict with a prohibition constraint, such as an SoD constraint. Consider the following delegation rule and SSD constraint, which has been defined in the aforementioned RBAC policy:

`DelegationConstraintAM:can_delegate(accountingManager,teller,1).`

`SSD-A-T`: a pair of roles (`accountant`, `teller`) is conflicting.

`DelegationConstraintAM` claims that a user who is a member of the role `teller` can be assigned to the delegated role `accountingManager` or its junior role `accountant`. As explained in Section 5, the class `Delegation-Authorization` expresses the *can_delegate* relation. In the object diagram of Figure 13, we see the object `auth_AM_T`, which represents `Delegation-ConstraintAM`. Please note that the delegation is only possible if the delegated user has the role `teller`, which is represented by the link between `auth_AM_T` and the role `teller`. Due to the fact that the maximum delegation is set to '1' by `DelegationConstraintAM`, the attribute `maxDelegationDepth` of the delegated role `accountingManager` must also be set to '1'.

The constraint `SSD-A-T` defines that the roles `accountant` and `teller` conflict with each other and cannot be assigned to the same user simultaneously. Thus, `DelegationConstraintAM` is in conflict with the `SSD-A-T`

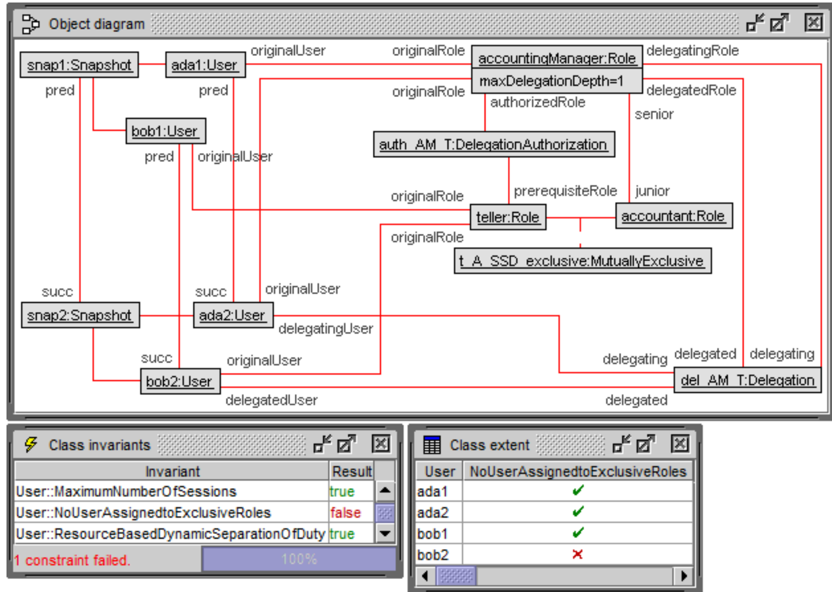

Figure 13: Delegation scenario with a conflict between a delegation and an SSD constraint.

constraint. The scenario (film strip) depicted in Figure 13 shows a situation which reveals this problem. Here, Ada delegates role `teller` to Bob. The temporal aspect of this delegation can be shown with the help of the two snapshots `snap1` and `snap2` which represent two successive points in time, namely, before and after delegation. In the second point in time, Bob has received the role `accountingManager` (see the object `del_AM_T` of type `Delegation`).

Employing USE, we can see that the invariant `User::AssignedtoExclusiveRoles` is violated by user Bob in the second point in time (`bob2` represents user Bob in snapshot `snap2`). This problem stems from the fact that Bob has indirectly received the role `accountant` through the delegation of the `accountingManager` role, i.e., the delegation and the prohibition constraints conflict with each other. In summary, an administrator can predefine such test cases as object diagrams and can thereafter test the policy with these test cases employing the USE tool.

Looking again at Figure 13, one can see that the displayed object diagram represents both the policy and the user access level. Specifically, the policy part covers delegation authorization, e.g., the delegation authorization object `auth_AM_T` and the attribute `maxDelegationDepth` of the account-

ingManager role. An administrator only needs to create objects and links as well as set attributes for these instances. She is not directly confronted with OCL constraints (as given in the sections 3.2.1 and 5), which again underlines the DSL aspect of our approach. The user access level (which an administrator can utilize for testing purposes) encompasses the concrete delegation steps. Similar remarks hold for the other scenarios described below. In the following, we give the second scenario discussing the revocation concepts as well as undesirable properties of revocation constraints.

6.3.2. Scenario 2: Blocked Access through Undesirable Revocation

In the following, we describe a situation in which an access is blocked due to revocation. First, we define the following delegation and revocation constraints:

`DelegationConstraintAM`: `can_delegate(accountingManager, , 1).`

`RevocationConstraintAM`: `can_revokeGDStrongCasc(accountingManager).`

The revocation is grant-dependent, strong, and cascading. Furthermore, we assume that user Ada has the roles `accountant` and `accountingManager` via original user assignment. Figure 14 now depicts a situation, which consists of four points in time (snapshots) and which is allowed by these rules and the RBAC policy defined above:

1. Snapshot 1: Ada delegates role `accountant` to Cyd by means of the `accountingManager` role. This is expressed by the delegation `del_A_T`.
2. Snapshot 2: Ada delegates role `accountingManager` to Cyd. This is expressed by the delegation `del_AM_T`.
3. Snapshot 3: Cyd delegates role `accountant` to Dan via the role `accountingManager` delegated in the step before. This is expressed by the delegation `delDel_AM_T`.
4. Snapshot 4: Dan tries to execute the action `createLedgerReport` on the resource `legderReport1`. At the same time, Ada revokes the role `accountant` from Cyd. This revocation is grant-dependent, strong, and cascading.

With the help of USE, we now can learn that the invariant `Session::ActionsPermitted` is violated, i.e., Dan is not permitted to execute the action

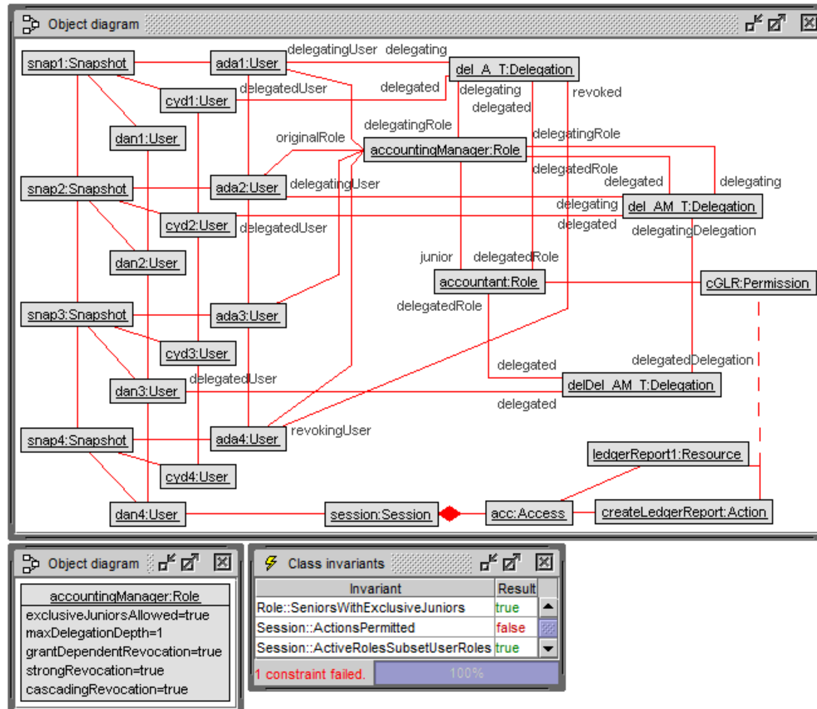

Figure 14: Revocation scenario blocking access to resources.

`createLedgerReport` on the resource `legderReport1` even though the permission has not been directly revoked from him. Due to the strong revocation, Cyd also loses the `accountingManager` role in addition to the `accountant` role. As a consequence, however, Dan also loses the `accountant` role because this role has been delegated via the role `accountingManager` and all delegations based on `accountingManager` are revoked cascadingly (see the small object diagram, which shows the configuration with respect to the `accountingManager` role, at the bottom of Figure 14). Without the role `accountant`, Dan does not have the permission to execute `createLedgerReport` on `ledgerReport1` anymore.

Again, an administrator can use such a scenario (film strip) as a test case for the policy validation with USE, specifically, testing complex access control concepts, such as cascading and strong revocation. Furthermore, the object diagram depicted in Figure 14 shows that we store the whole "access control state". This means, even if delegations are revoked, the `Delegation` objects and the corresponding links are not deleted. If our metamodel for role-based

delegation is used as a basis for the implementation of an authorization engine [27, 35], this information can be used for an audit trail.

6.3.3. Scenario 3: Identify Leaking Permissions

The scenarios discussed presume that the administrator already knows the situations in detail which might go wrong. Sometimes, however, an administrator wishes to let the validation tool generate automatically desirable or undesirable scenarios. In this case, the USE model validator comes into play. Please note that we currently cannot handle revocation policies here because Kodkod is based on relational logic and hence recursion, which is used for the definition of the OCL operation `isRevoked())` (see Section 5), cannot be expressed.

Again, the basis of this scenario is the aforementioned RBAC policy, but with the exception that we define the following DSD constraint for the roles `teller` and `accountant` instead of the SSD constraint mentioned above:

`DSD-Role-A-T`: the pair of roles (`accountant`, `teller`) is conflicting w.r.t. to role activation within a session.

This constraint may have been introduced by an administrator for the sake of a higher flexibility compared to the SSD constraint (less strict policy). In addition, the following delegation constraint is defined by the administrator:

```
DelegationConstraintT: can_delegate(teller, , 1).
```

Furthermore, the following user assignment relations are presumed:

```
UAO(Ada,accountant), UAO(Bob,customerServiceRep),
UAO(Cyd,teller), UAO(Dan,teller)
```

Now, the administrator would like to test this new policy with the USE model validator. She first defines a property which she expects to be satisfied by the policy. Typically, this can be a safety property. The safety problem is well-known in access control literature and states whether access rights can leak to a user [36]. In our case, she might ask whether a user can execute the actions `inputDepositAccount` and `createLedgerReport` on resources. This situation should not be possible because these actions are assigned to the mutually exclusive roles `teller` and `accountant`, respectively.

We can formulate the aforementioned property in OCL as follows:

```
User.allInstances()->exists(u|
 u.session.access.action->includesAll(Set{inputDepositAccount,
                                          createLedgerReport}))
```

We now give this property, the RBAC policy, and the delegation policy to the USE model validator. The search space for the USE model validator then is defined by:

1. the RBAC/delegation policy as a partial solution,
2. the additional property,
3. all UML and OCL constraints.

For the classes `Session`, `Access`, `Delegation`, and `Snapshot`, we only configure a maximum number *max* of instances (objects) rather than define concrete instances in advance. All possible attribute value and link combinations are considered for the attributes and associations belonging to the aforementioned classes. In particular, if we consider the class `Delegation`, this means that the USE model validator generates delegation steps (instances of the class `Delegation`) on its own. Starting from $max = 0$, we can successively increment *max* for each class until we obtain a solution (or we are confident that no solution exists). Proceeding this way, the search space is gradually increased.

In Figure 15, we see the solution which satisfies both the query and the policy and which is an undesirable system state. Ada can access both actions, although this should have been prohibited by the DSD constraint. Ada obtains the `teller` role from Dan (in addition to the `accontant` role). This delegation step is not forbidden because we have not defined an *SSD constraint* between `accontant` and `teller`. Since the DSD rule only considers one session, Ada now can activate both roles in two different sessions `session1` and `session2` and finally execute the actions `inputDepositAccount` and `createLedgerReport`.

In summary, this example shows that USE model validator can be effectively used in testing RBAC/delegation policies. In addition, the USE model validator generated the solution depicted in Figure 15 only within a few seconds on an ordinary laptop. Improving on the user interface in the future, we believe that our validation environment consisting of USE and the model validator can be employed in organizations to test real-world policies.

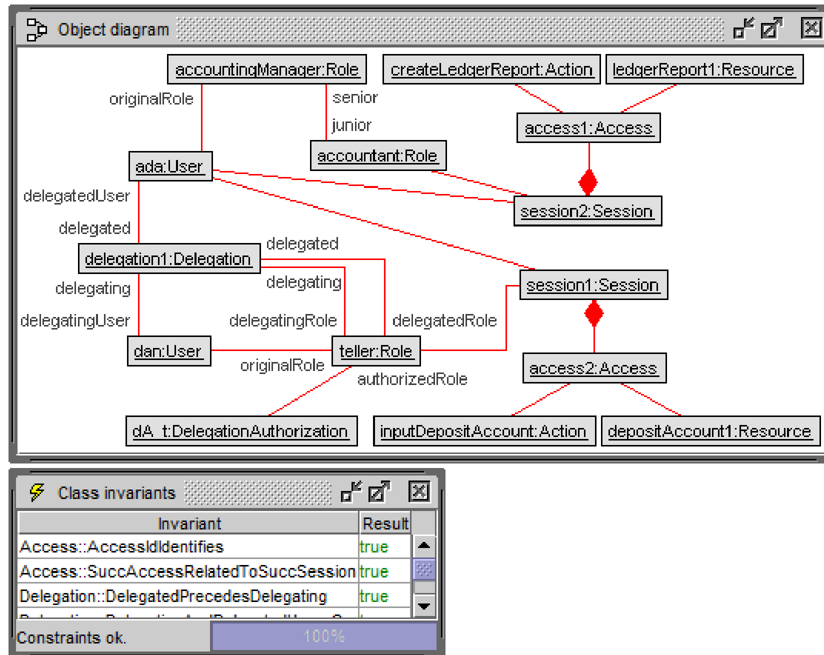

Figure 15: A system state with an undesirable situation, automatically generated by the USE model validator.

7. Related Work

In this section, we will show that our RBAC DSL is the first UML-based specification language which supports delegation and revocation schemes as well as advanced RBAC concepts, such as history-based SoD, a concept completely different to the simple authorization constraints, which are enumerated in the RBAC ANSI standard [23].

There is a plethora of works integrating security policies into system models based on UML. We have already commented on our earlier works [12, 18] in the introduction of this paper, in particular, we now support the RDM2000 model for delegation and revocation. Other works which discuss role-based policy languages based on UML include [28, 37, 38, 39, 40, 13]. Some of the approaches do not particularly address RBAC like UMLsec [37]. Basin et al. present the modeling language SecureUML for integrating the specification of access control into application models and for automatically generating access control infrastructures of applications [13]. They also deal with authorization constraints, but do not support SoD constraints. Furthermore,

SecureUML is based on the UML profile mechanism and hence it is unclear whether and to which extent current CASE tools support UML profiles, whereas we use UML object diagrams, a very basic diagram type.

In [28], we explicitly model role-based SoD constraints with UML and OCL. There, we have no means for handling dynamic aspects and we do not strictly separate the presented RBAC metamodel from concrete policy definitions. Ray et al. [38] solve the latter problem by generically designing the authorization constraints. We follow their approach with respect to the RBAC description presented in this paper and extend it in terms of dynamic aspects as well as delegation and revocation. Owing to the fact that Ray et al. utilize a template mechanism, only those UML object diagrams can be expressed for which templates have been defined. Consequently, the expressiveness of their policy language is more restricted than ours.

There is also recent work by Strembeck and Mendling with a similar goal by providing a DSL which hides the OCL constraints. In particular, the authors express role-based policies for business processes by an extension of the UML2 metamodel (extended UML activity diagrams) [41]. As a consequence, UML CASE tools do not support this extension such that the authors had to develop their own tool support for processing their DSL in contrast to our approach based on object diagrams. Moreover, delegation and revocation is not treated in this work as directly noted in the conclusion. Conversely, our RBAC DSL currently does not cover business processes, which remains interesting future work. We believe that the snapshot models provide a good foundation for this task.

Several works on the validation of RBAC policies based on UML and OCL have been presented [42, 30, 28, 43]. Based upon SecureUML, Basin et al. propose an approach to analyzing RBAC policies by stating and evaluating queries like 'Which permissions can a user perform with a given role?' or 'Are there two roles with the same set of permissions?' [42]. Although not explicitly addressed in this paper, our approach allows the same kind of queries through the query facility of the USE tool [15] into which the model validator is integrated. In [30], a scenario-based approach to analyzing UML models is presented which is exemplified by an elementary RBAC UML model. In this context, a policy is considered as a dynamic artifact which evolves through administrator activities. Hence, it can be examined whether a sequence of administrative RBAC operations such as assigning users to roles can violate static SoD constraints. In contrast, we realize dynamics at the end-user level, enabling dynamic SoD. Administrative actions are implicitly involved in our

approach when analyzing partial policies. In addition, our RBAC metamodel consists of both a static and a dynamic part.

The main difference between all the aforementioned and our current work lies in the fact that we now support delegation and revocation based upon a concrete and well-established delegation and revocation model [7]. None of the earlier works on RBAC and UML has tackled the problem of delegation and different revocation semantics (as given by Hagström et al.) before.

There exist some tools designed to support the analysis of general access control systems. In particular, the SERSCIS Access Modeller (SAM) [44], which is inspired by Scollar [45], takes a model of a system and strives to validate certain security properties about the system via examining all the ways access can propagate through the system. However, this tool can only model an RBAC system with limited notations and relations. In contrast, our approach could represent complex RBAC systems with advanced concepts like delegation and revocation. Besides, our DSL has the capability to express a wide range of policies including history-based SoD and various delegation and revocation schemes. In addition, due to the fact that we leverage basic UML notations for model representation, our DSL can be processed by most existing UML tools.

There are other approaches to the formal specification of access control policies with notions of delegation, notably the work by Becker et al. [46]. In particular, they designed the SecPAL language, which also hides the technical (and formal) details behind a DSL. In contrast to the SecPAL approach, we can exploit the rich tool support available for UML and OCL (CASE and validation tools).

8. Conclusion and Outlook

In this paper, we presented a UML-based graphical DSL for role-based delegation and revocation. In particular, our DSL allows an administrator to define role-based access control policies with complex concepts, such as different revocation schemes, in UML object diagrams. This hides the complexity inherent in OCL. Moreover, we showed how to validate delegation and revocation with the USE tool and the USE model validator. This allows an administrator to identify subtle security holes, induced by the interplay between delegation and revocation rules with other advanced access control concepts, such as role hierarchies and authorization constraints.

There is plenty of room for future work. First, we can improve our user interface. For example, currently we can only query properties of role-based policies with the help of OCL queries. For often recurring queries, specific user interfaces can be made available. Also, a graphical DSL for role-based policies can be designed, which uses specific language constructs for access control, such as the graphical language proposed by Jaeger and Tidswell [26]. Having a transformation between our UML-based DSL and the specific DSL at hand, validation tools such as USE and the USE model validator can still be used. Furthermore, we can develop an authorization engine, which enforces the delegation and revocation policies and can be integrated with IT infrastructures of organizations. Last but not least, we can apply our approach to other domains, such as the healthcare domain to express policies on electronic health records, and perform larger case studies to evaluate usability and effectiveness of our proposal.

References

[1] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, Role-Based Access Control Models, IEEE Computer 29 (2) (1996) 38–47.

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

[3] 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 (SSP ’98), IEEE, 1998, pp. 172–185.

[4] C. Georgiadis, I. Mavridis, G. Pangalos, R. Thomas, Flexible team-based access control using contexts, in: Proc. of the ACM Symposium on Access Control Models and Technologies, 2001, pp. 21–27.

[5] J. Joshi, E. Bertino, U. Latif, A. Ghafoor, A generalized temporal role-based access control model., IEEE Trans. Knowl. Data Eng. 17 (1) (2005) 4–23.

[6] M. J. Nash, K. R. Poland, Some conundrums concerning separation of duty, in: Proc. IEEE Symposium on Research in Security and Privacy, 1990, pp. 201–207.

[7] L. Zhang, G.-J. Ahn, B.-T. Chu, A rule-based framework for role-based delegation and revocation, ACM Transactions on Information and System Security 6 (3) (2003) 404–441.

[8] J. Joshi, E. Bertino, Fine-grained role-based delegation in presence of the hybrid role hierarchy, in: Proc. of the 11th ACM Symposium on Access Control Models and Technologies, Lake Tahoe, California, USA, 2006, pp. 81–90.

[9] E. Barka, R. Sandhu, A role-based delegation model and some extensions, in: Proc. of 16th Annual Computer Security Application Conference, 2000, pp. 125–134.

[10] J. Wainer, A. Kumar, A fine-grained, controllable, user-to-user delegation method in RBAC, in: Proc. of the 10th ACM Symposium on Access Control Models and Technologies, Stockholm, Sweden, 2005, pp. 59–66.

[11] V. Atluri, J. Warner, Supporting conditional delegation in secure workflow management systems, in: Proc. of the 10th ACM Symposium on Access Control Models and Technologies, Stockholm, Sweden, 2005, pp. 49–58.

[12] M. Kuhlmann, K. Sohr, M. Gogolla, Comprehensive Two-level Analysis of Static and Dynamic RBAC Constraints with UML and OCL, in: Fifth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011, IEEE Computer Society, 2011, pp. 108–117.

[13] D. A. Basin, J. Doser, T. Lodderstedt, Model driven security: From UML models to access control infrastructures, ACM Trans. Softw. Eng. Methodoly 15 (1) (2006) 39–91.

[14] M. Kuhlmann, L. Hamann, M. Gogolla, Extensive validation of OCL models by integrating SAT solving into USE, in: J. Bishop, A. Vallecillo (Eds.), Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings, Vol. 6705 of Lecture Notes in Computer Science, Springer, 2011, pp. 290–306.

[15] M. Gogolla, F. Büttner, M. Richters, USE: A UML-Based Specification Environment for Validating UML and OCL, Science of Computer Programming 69 (2007) 27–34.

[16] A. Hagström, S. Jajodia, F. Parisi-Presicce, D. Wijesekera, Revocations – a classification, in: 14th IEEE Computer Security Foundations Workshop (CSFW '01), 2001, pp. 44–58.

[17] R. Chandramouli, Application of XML tools for enterprise-wide RBAC implementation tasks, in: Proceedings of the fifth ACM workshop on Role-based access control, Berlin, Germany, 2000, pp. 11–18.

[18] M. Kuhlmann, K. Sohr, M. Gogolla, Employing UML and OCL for Designing and Analyzing Role-Based Access Control, Mathematical Structures in Computer Science. To appear.

[19] Object Management Group, OMG Unified Modeling Language (OMG UML), Infrastructure - Version 2.3, `formal/2010-05-03` (May 2010).

[20] Object Management Group, OMG Unified Modeling Language (OMG UML), Superstructure - Version 2.3, `formal/2010-05-03` (May 2010).

[21] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, 2nd Edition, Object Technology Series, Addison-Wesley Professional, Boston, Massachusetts, 2004.

[22] Object Management Group, Object Constraint Language - Version 2.2, `formal/2010-02-01` (Feb. 2010).

[23] American National Standards Institute Inc., Role Based Access Control, ANSI-INCITS 359-2004 (2004).

[24] X. Zhang, S. Oh, R. Sandhu, Pbdm: a flexible delegation model in rbac, in: Proceedings of the eighth ACM symposium on Access control models and technologies, ACM, 2003, pp. 149–157.

[25] J. Crampton, H. Khambhammettu, Delegation in role-based access control, International Journal of Information Security 7 (2) (2008) 123–136.

[26] T. Jaeger, J. Tidswell, Practical safety in flexible access control models, ACM TISSEC 4 (2) (2001) 158–190.

[27] K. Sohr, T. Mustafa, X. Bao, G.-J. Ahn, Enforcing Role-Based Access Control Policies in Web Services with UML and OCL, in: Proceedings of the 23th Annual Computer Security Applications Conference, IEEE Computer Society, 2008, pp. 257–266.

[28] K. Sohr, M. Drouineaud, G.-J. Ahn, M. Gogolla, Analyzing and Managing Role-Based Access Control Policies, IEEE Trans. Knowl. Data Eng 20 (7) (2008) 924–939.

[29] M. Kuhlmann, M. Gogolla, Modeling and Validating Mondex Scenarios Described in UML and OCL with USE, Formal Aspects of Computing 20 (1) (2008) 79–100.

[30] L. Yu, R. B. France, I. Ray, Scenario-Based Static Analysis of UML Class Models, in: Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Vol. 5301 of LNCS, Springer, Berlin, 2008, pp. 234–248.

[31] M. Kuhlmann, K. Sohr, M. Gogolla, RBAC Metamodel: Sources and Validation Results, `http://www.db.informatik.uni-bremen.de/publications/Kuhlmann_2010_RBAC_ sources.pdf` (2010).

[32] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, UML2Alloy: A Challenging Model Transformation, in: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Vol. 4735 of LNCS, Springer, Berlin, 2007, pp. 436–450.

[33] M. Gogolla, M. Kuhlmann, L. Hamann, Consistency, Independence and Consequences in UML and OCL Models, in: Proc. 3rd Int. Conf. Test and Proof (TAP'2009), Springer, Berlin, LNCS 5668, 2009, pp. 90–104.

[34] M. Kuhlmann, K. Sohr, M. Gogolla, H. Hu, G.-J. Ahn, USE specifications of the metamodel for role-based delegation and revocation, `http://www.db.informatik.uni-bremen.de/publications/RDM2000_metamodel.use` (2011).

[35] M. Zurko, R. Simon, T. Sanfilippo, A user-centered, modular authorization service built on an RBAC foundation, in: Proc. of the IEEE Symposium on Research in Security and Privacy, Oakland, CA, 1999, pp. 57–71.

[36] M. S. Harrison, W. L. Ruzzo, J. D. Ullman, Protection in operating systems, Communications of the ACM 19 (8).

[37] J. Jürjens, UMLsec: Extending UML for secure systems development, Lecture Notes in Computer Science 2460 (2002) 412–425.

[38] I. Ray, N. Li, R. B. 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, ACM Press New York, USA, 2004, pp. 115–124.

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

[40] E. Fernández-Medina, M. Piattini, Extending OCL for secure database development, in: Proc. of UML 2004 - The Unified Modeling Language: Modeling Languages and Applications, Vol. 3273 of LNCS, Springer, 2004, pp. 380–394.

[41] M. Strembeck, J. Mendling, Modeling process-related RBAC models with extended UML activity models, Information and Software Technology 53 (5) (2011) 456 – 483.

[42] D. A. Basin, M. Clavel, J. Doser, M. Egea, Automated analysis of security-design models, Information & Software Technology 51 (5) (2009) 815–831.

[43] S. Höhn, J. Jürjens, Automated checking of SAP security permissions, in: 6th Working Conference on Integrity and Internal Control in Information Systems (IICIS), Kluwer, Lausanne, Switzerland, 2003.

[44] SERSCIS Access Modeller, `http://www.serscis.eu/sam/`.

[45] The Scoll Project, `http://www.scoll.evoluware.eu/`.

[46] M. Y. Becker, C. Fournet, A. D. Gordon, SecPAL: Design and semantics of a decentralized authorization language, Journal of Computer Security 18 (4) (2010) 619–665.