Under consideration for publication in Math. Struct. in Comp. Science

Employing UML and OCL for Designing and Analyzing Role-Based Access Control

MIRCO KUHLMANN[1], KARSTEN SOHR[2] and MARTIN GOGOLLA[1]

[1] *Computer Science Department, Database Systems Group, D-28334 Bremen, Germany.*
[2] *Center for Computing Technologies, D-28334 Bremen, Germany.*

Received 25 August 2011

Stringent security requirements of organizations like banks or hospitals frequently adopt role-based access control (RBAC) principles to represent and simplify their internal permission management. While representing a fundamental advanced RBAC concept enabling precise restrictions on access rights, authorization constraints increase the complexity of the resulting security policies so that tool support for comfortable creation and adequate validation is required. One contribution of our work is a new approach to developing and analyzing RBAC policies using a UML-based domain-specific language (DSL), which allows hiding the mathematical structures of the underlying authorization constraints implemented in OCL. The presented DSL is highly configurable and extensible with respect to new concepts and classes of authorization constraints, and allows the developer to validate RBAC policies in an effective way. The handling of dynamic (i. e., time-dependent) constraints, their visual representation through the RBAC DSL, and their analysis form another part of our contribution. The approach is supported by a UML and OCL validation tool.

1. Introduction

Role-based access control (RBAC) provides a framework for managing permissions respecting access to many kinds of resources. Today, RBAC is an established standard and is used in many areas with stringent security demands. One of the main advantages of RBAC is that high-level organizational rules can be implemented in a natural way (Sandhu et al., 1996). The basic RBAC concepts represent a simple configuration of users, roles and permissions, as well as user and permission assignments to roles. In order to address practical requirements, RBAC has been extended with respect to more advanced concepts like role delegation, role hierarchies or formal authorization constraints.

In particular, those advanced RBAC concepts are an important means for laying out higher-level organizational rules. Typical rules enforce separation of duty (SoD). As pointed out in the literature (Nash and Poland, 1990; Simon and Zurko, 1997), history-based SoD is a flexible form of SoD which is often needed in practice. For example, in a banking application, a clerk may have the permissions to prepare and authorize cheques, but once the clerk has prepared a cheque, she cannot authorize it any more. Various

other types of dynamic (in particular application-specific) authorization constraints have also been identified in prior work (Bertino et al., 1999; Schaad et al., 2006).

Usually, RBAC rules become complex in large organizations such as financial institutes or hospitals so that undesirable properties of the security policies, i. e., the specific role configurations and sets of authorization constraints, may arise. For example, an SoD constraint between two roles may be useless if a user can obtain the security-critical permissions through other roles with no SoD restrictions (Li et al., 2007). Therefore, comprehensive RBAC policies need to be thoroughly analyzed to ensure a correct realization of the underlying requirements. Dynamic (i. e., stateful) properties are of special interest because many important constraints like history-based SoD regard both present as well as past or future activities so that specific sequences of resource accesses must be forbidden, e.g., after a preparation of a cheque by a clerk its authorization by the same clerk is not allowed. The relevance of stateful access control has often been pointed out in literature (Nash and Poland, 1990; Simon and Zurko, 1997; Bertino et al., 1999; Barth et al., 2006; Clark and Wilson, 1987; Sandhu, 1988).

The Unified Modeling Language (UML) in combination with the Object Constraint Language (OCL) provides a promising way for creating and analyzing RBAC policies, since both languages benefit from substantial tool support including model-driven methods. We present a UML description of the core RBAC concepts and supplemental authorization constraints representing an RBAC metamodel. The metamodel is meant as a basis for conceptual explanations within this paper. But even this simple model allows administrators (i. e., security officers) entrusted with the permission management to specify and examine policies with respect to explicit and implicit static and dynamic properties. After a comprehensive validation, a policy can be deployed and enforced by an authorization engine (Sohr et al., 2008b). Our RBAC metamodel includes typical SoD constraints. However, it can be enriched by a broad variety of dynamic access control options including constraints specifically designed for a particular organization.

As will be explained below, we achieve the handling of dynamic constraints on the technical side by introducing snapshots (i. e., concrete states of the modeled system) together with particular temporal relationships (predecessor and successor relationships) forming sequences of system states which are called scenarios. Thereby, we enable the expression of dynamic constraints which cannot be handled in other approaches based on UML and OCL like (Yu et al., 2008). See Sect. 6 for further discussion.

With our RBAC metamodel, we present a domain-specific language (DSL) adapted for RBAC, syntactically based on UML class and object diagrams. It assists administrators in configuring complex policies, while hiding the details of the underlying mathematical structures (like the OCL invariants), which realize the RBAC authorization constraints. We employ axiomatic specifications, i. e., our basic mathematical structures are determined by signatures and axioms. We denote signatures by UML class diagrams and axioms by OCL constraints. An interpretation for a signature is given in our approach by UML object diagrams. The subset of UML and the part of OCL which we use have a precise mathematical set-theoretic semantics (Richters and Gogolla, 2001). The administrators design their policies by designing object diagrams, i.e., they create objects and links and define attribute values. At this 'policy level', the administrators are allowed

to enforce complex formal constraints without the need to textually define or adjust the respective mathematical formulas. These complex constraints can be configured through specific attributes and links. A single constraint is formulated only once, and becomes active when its configuration context is settled by an administrator. Also, examinations at the so-called 'user access level' can be carried out in relation to the defined policies without knowing the internal matters of the RBAC metamodel. For validation purposes, user activities can be simulated at the user access level in the context of a given policy. In hiding the complexity, we see a great advantage compared to formal approaches based on methods that require handling of complex textual structures (like XACML (Abi Haidar et al., 2006)) during the policy specification and validation process. In an organizational context, a small number of UML and OCL experts can maintain and extend the RBAC DSL providing the administrators with the necessary functionality.

The DSL supports the constructs needed for a formal analysis of the created policy. These constructs including dynamic authorization constraints can be directly handled by any tool allowing for validation of UML models because all temporal requirements are encoded within an ordinary UML class diagram and OCL invariants. More generally, our approach for handling dynamics can be applied to other access control models and even to usage control models which comprise temporal aspects (reactive systems) (Barth et al., 2006; Dougherty et al., 2006; Hilty et al., 2007; Zhang et al., 2005).

Development of the RBAC UML metamodel, as well as the development of its instances (i.e., the organizational RBAC policies), is a time-consuming task, involving comprehensive analysis and evaluation. This requires a powerful UML and OCL validation tool. An advantageous way for automatically analyzing complex UML models with OCL constraints is the use of our newly developed SAT-based 'model validator' (Kuhlmann et al., 2011a), which is implemented as a plug-in for the UML-based Specification Environment (USE) (Gogolla et al., 2007). The validator makes use of the relational model finder Kodkod (Torlak and Jackson, 2007), which, in turn, makes use of solvers for boolean satisfiability (SAT).

In the following sections, we will consider and analyze our RBAC description from different perspectives, i. e., at different levels, in order to give a deeper insight into the RBAC metamodel and the given possibilities. The new USE model validator allows us to realize this approach, as it efficiently executes our validation demands. Its application is not limited to the RBAC context, however. It can be applied to other UML and OCL models as well.

In summary, we combine the following aspects with the development and analysis of RBAC policies:

— Support for dynamics through snapshot modeling, enabling the tracing of user activities over time and the expression of dynamic constraints.
— Separation of the RBAC metamodel from concrete policy definitions.
— Hiding the complexity of the authorization constraints from the policy developers (DSL approach).
— Flexibility to extend the RBAC DSL with respect to specific organizational demands.
— Supporting automated analysis of RBAC policies.

This paper is based on our work presented in (Kuhlmann et al., 2011b). The rest of the

paper is structured as follows. In Sect. 2 we introduce the basic concepts of RBAC, UML, OCL, and DSLs, as well as the USE system. Section 3 presents the different levels in the RBAC UML description. The RBAC metamodel itself is explained in Sect. 3.1 and the supplemental OCL constraints in Sect. 3.2. We address the different levels of analysis with respect to the RBAC UML description in Sect. 4. In Sect. 4.1 we briefly introduce the USE model validator, before we discuss the analysis of the RBAC metamodel in Sect. 4.2 and RBAC policies in Sect. 4.3. A detailed case study, which applies the RBAC DSL and illustrates particular extensions to the metamodel, is discussed in Sect. 5. Section 6 presents related work. We conclude with Sect. 7.

2. Introduction of the Employed Approaches

Within this section, we provide background information on the main concepts of the modeling languages UML and OCL, domain-specific languages and RBAC. We also sketch the purpose and features of the UML tool USE.

2.1. *RBAC 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 (Sandhu et al., 1996). We now give an overview of (general) hierarchical RBAC according to the RBAC standard (American National Standards Institute Inc., 2004), which is the basis of our 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 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

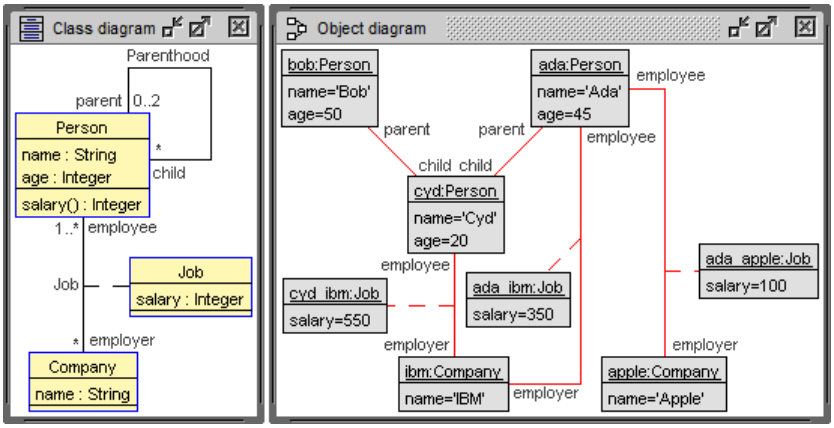

Fig. 1. Example UML class and object diagram

the *cashier* and *cashier supervisor* role, i.e., the UA relation is restricted. It has been argued elsewhere (Sandhu et al., 1996) 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 consider static and dynamic SoD (Gligor et al., 1998; Simon and Zurko, 1997) and cardinality constraints (Sandhu et al., 1996). Temporal considerations need extra preparation which we introduce later.

2.2. *The Unified Modeling Language*

The Unified Modeling Language (UML) (Object Management Group, 2010b; Object Management Group, 2010c) 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 (Rumbaugh et al., 2004). 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.

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.

2.3. *The Object Constraint Language*

The Object Constraint Language (OCL) (Object Management Group, 2010a) 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 Fig. 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)
```

2.4. *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 Fig. 2 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

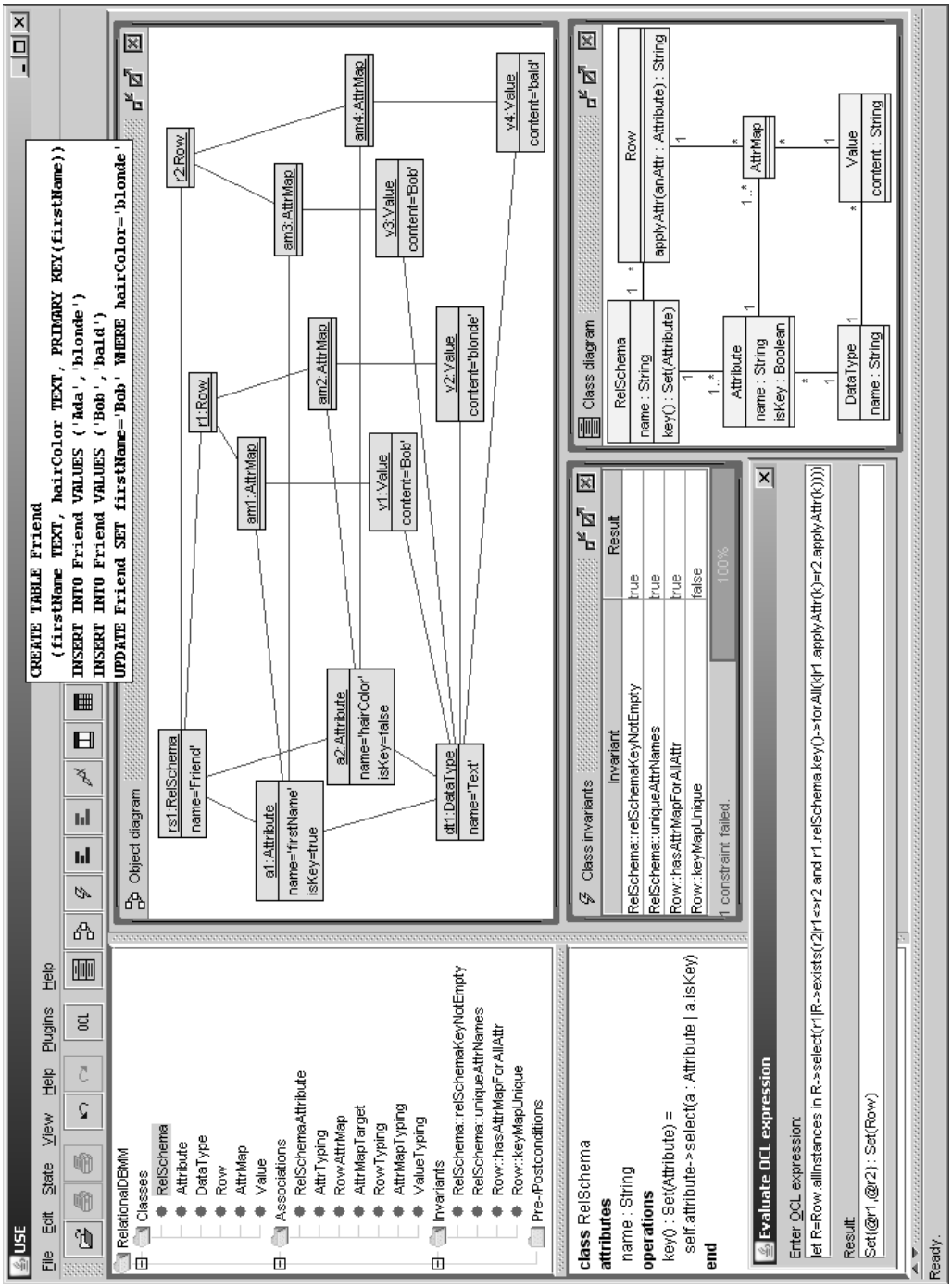

Fig. 2. USE screenshot of Relational DB Metamodel

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 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 set `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 deleted 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.

2.5. *The USE Tool*

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

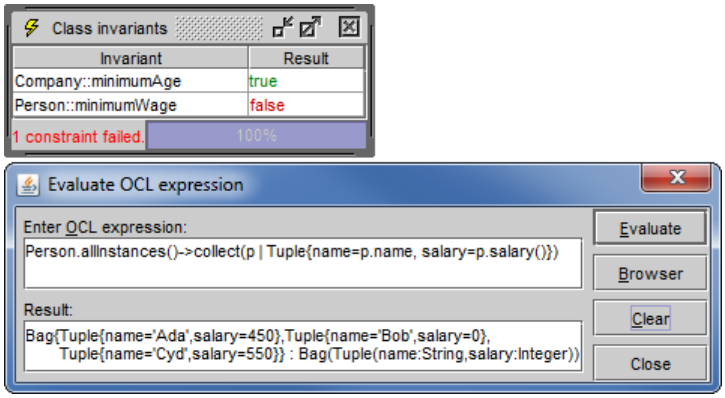

Fig. 3. Evaluation of class invariants and a user-defined OCL query expression in USE

model instances. The central idea 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 Fig. 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 Fig. 3). 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 Fig. 3. 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.

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*

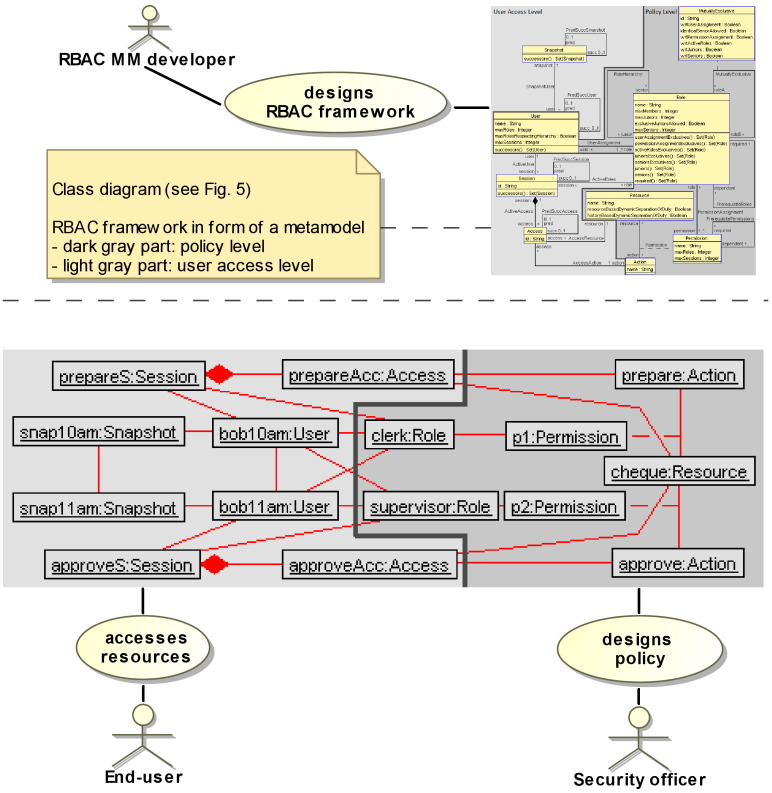

Fig. 4. Policy and user access level of the RBAC UML description

level for the policy design and a *user access level* for the policy analysis. Figure 4 visualizes the basic idea. An object diagram shows an example instance of the RBAC class diagram. The dark grey part represents a rudimentary 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. 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) cheque. (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 cheques. 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 cheque (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 Fig. 4 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 cheque at 10 am and approves this cheque in a different session at 11 am, thus, violating the SoD requirement. Speaking more precisely, bob accesses the real resource 'cheque' 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*.

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 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 'cheque', 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 ways to analyze the RBAC description.

3.1. *RBAC Metamodel*

The object diagram shown in Fig. 4 is based on the RBAC metamodel shown in Fig. 5. Classes and associations belong analogously to the policy level or the user access level.

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 (Sandhu et al., 1996) supplemented by dynamic constraints discussed in (Sohr et al., 2008a). In our approach, the constraints are realized as UML attributes and associations.

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

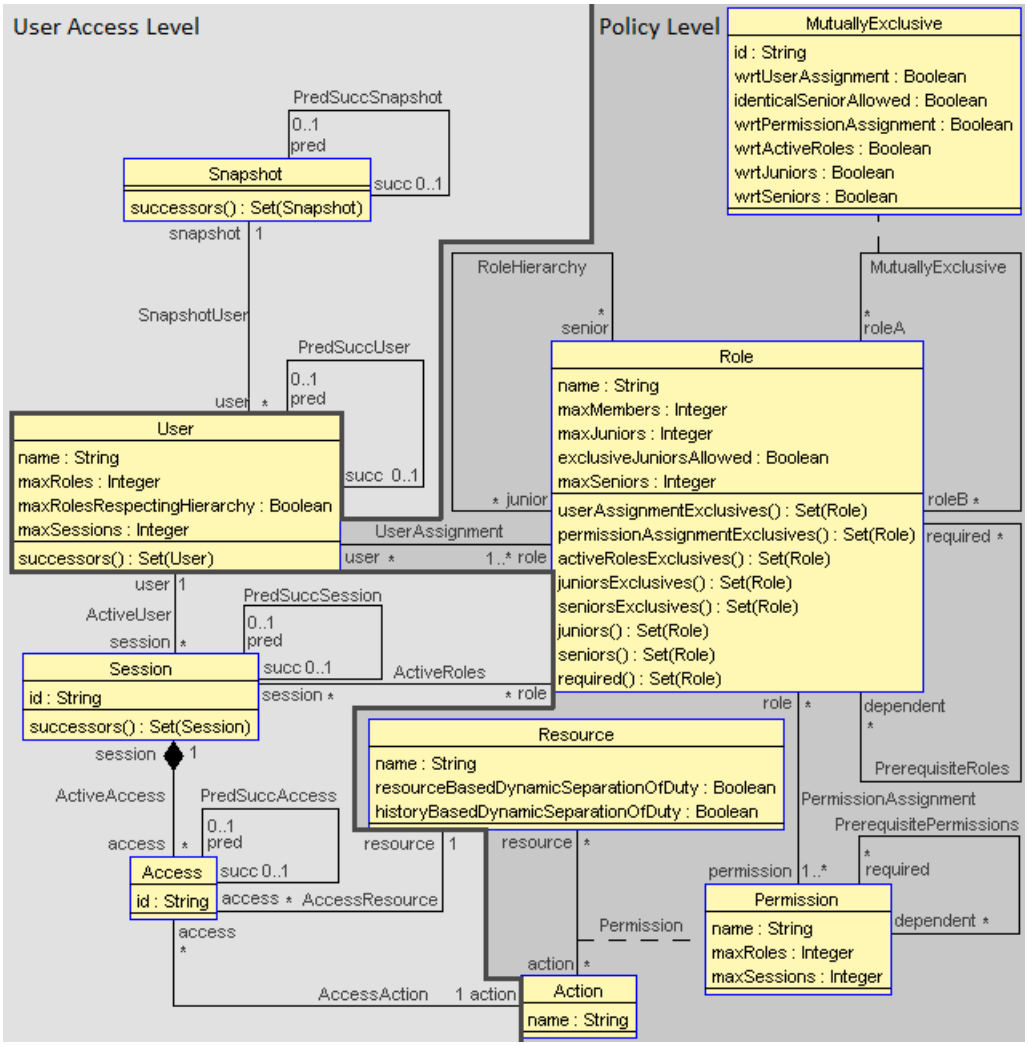

Fig. 5. The RBAC metamodel

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 Fig. 5 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 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

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')	(Sandhu et al., 1996), page 11, lines 29–30
	`maxSessions`	maximum number of simultaneously active sessions with respect to a user	(Sandhu et al., 1996), page 12, lines 15–16
Role::	`maxMembers`	maximum number of assigned users	(Sandhu et al., 1996), 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')	(Sandhu et al., 1996), page 12, lines 30–31
	`maxSeniors`	maximum number of senior roles	(Sandhu et al., 1996), page 12, lines 30–31
	`PrerequisiteRoles` (Assoc.)	dependent role postulates required role with respect to user assignment	(Sandhu et al., 1996), 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)	(Sandhu et al., 1996), page 11, lines 6–7
	`wrtPermissionAssignment`	a permission must not be assigned to both roles	(Sandhu et al., 1996), page 11, lines 10–12
	`wrtActiveRoles`	the connected roles must not be both activated in a session (possibly involving several snapshots)	(Sandhu et al., 1996), page 12, lines 14–15
	`wrtJuniors`	the connected roles must not have the same junior roles	(Sandhu et al., 1996), page 12, lines 31–32
	`wrtSeniors`	the connected roles must not have the same senior roles	(Sandhu et al., 1996), page 12, lines 31–32
Permission::	`maxRoles`	maximum number of roles the permission is assigned to	(Sandhu et al., 1996), page 11, lines 30–32
	`maxSessions`	maximum number of sessions simultaneously activating the permission (i. e., within the same snapshot)	(Sandhu et al., 1996), page 12, lines 16–17
	`PrerequisitePermissions` (Assoc.)	assignment of the dependent permission postulates the assignment of the required permission	(Sandhu et al., 1996), page 12, lines 1–3
Resource::			
	`resourceBasedDynamic-SeparationOfDuty`	a user may not apply more than one action to the resource	(Simon and Zurko, 1997), page 4, line 16–20
	`historyBasedDynamic-SeparationOfDuty`	a user may not apply all available actions to the resource	(Simon and Zurko, 1997), page 4, line 28–39

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* (Kuhlmann and Gogolla, 2008; Yu et al., 2008). That is, one object diagram for Fig. 5 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 'PredSucc' 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 (Yu et al., 2008).

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 Fig. 5). 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 constraint. 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. Further dynamic authorization constraints are discussed within Sect. 4.

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 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 (Kuhlmann et al., 2010).

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.

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

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**

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 UML-based Specification Environment (USE) (Gogolla et al., 2007). 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 (Sohr et al., 2008a), we employ the ASSL generator (Gogolla et al., 2007) 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 (Anastasakis et al., 2007). 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 (Kuhlmann et al., 2010) for details.

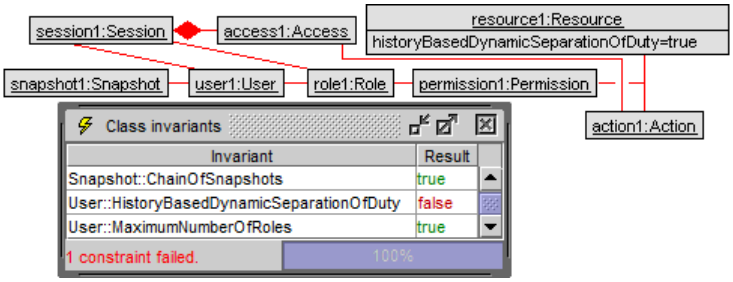

Fig. 6. Object diagram revealing an erroneous constraint definition

All examinations presented in the following sections complete within seconds using an ordinary laptop.

4.2. *Analyzing the RBAC Metamodel*

A comprehensive analysis of the RBAC metamodel during development helped us to discover several unwanted properties of which we present two as an example in this section. 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 (Gogolla et al., 2009).

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. This check has to be executed 30 times, as the RBAC metamodel currently comprises 30 OCL constraints. We automated the sequential checks with the model validator so that no further manual interaction is needed. 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 all constraints are independent. However, the consideration of the generated object diagrams is a part of a valuable analysis because they can reveal erroneous constraints. For example, the model validator returned the object diagram shown in Fig. 6 in an early development phase of the RBAC metamodel. As expected, the diagram proves the independence of the invariant 'HistoryBasedDynamicSeparationOfDuty' because all invariants but this are fulfilled. (The shown class invariant view of the USE tool displays an extract of all invariants.) However, the object diagram shows a situation which should normally not violate the respective invariant. The SoD constraint states that a user must not apply all available operations to a resource. But in this case, there was just one action which should be available for application. Thus,

the object diagram pointed out that we forgot to handle the particular case of exactly one available action. After correcting the invariant, we obtained an adequate, yet more complex result. The results for each invariant are presented in (Kuhlmann et al., 2010).

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. This way, we discovered a further erroneous invariant during development. We searched for an object diagram showing a simple policy with one resource and one action. The permission corresponding to the action–resource pair had to define a maximum number of active sessions (maxSessions = 1). Additionally, the diagram had to simulate a user access scenario with at least five snapshots and at least five sessions with user accesses. Although there should be many valid object diagrams, we got no solution with respect to this search space. After deactivating the invariants which had no effect on the result we found out that two invariants ('ActionsPermitted' and 'MaximumNumberOfSessions') were responsible for this unwanted behavior. The former invariant ensures that only permitted accesses to resources exist. The latter realizes the authorization constraint which controls the maximum number of sessions:

```
context p:Permission inv MaximumNumberOfSessions:
  p.maxSessions.isDefined implies
    p.role.session->asSet()->size() <= p.maxSessions
```

Within a session, permissions are indirectly activated by activating a role the permission is assigned to. Thus, the expression `p.role.session->asSet()->size()` returns the number of all (distinct) sessions which activated the current permission disregarding the time. That is, also closed sessions, which do not activate a permission any more, are involved. Hence, we adjusted the invariant to respect the dynamics resulting from the scenarios (see the constraint below). The maximum number of sessions is now calculated in the context of the individual snapshots so that only simultaneously active sessions are counted.

```
context p:Permission inv MaximumNumberOfSessions:
  p.maxSessions.isDefined implies
    Snapshot.allInstances()->forAll(snap |
      p.role.session->asSet()->select(s |
        s.user.snapshot = snap)->size() <= p.maxSessions)
```

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

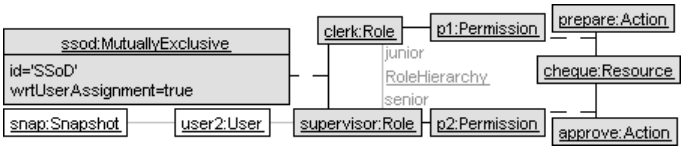

Fig. 7. Partial policy and partial search results (white objects, grey links)

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 '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 Fig. 7 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

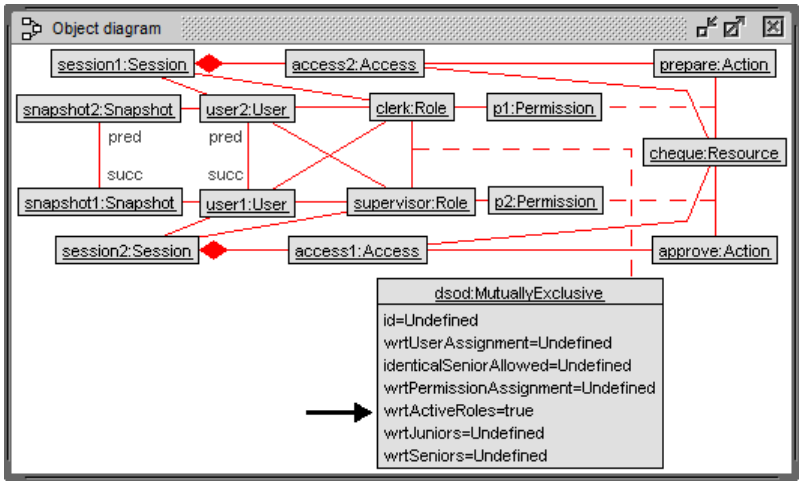

Fig. 8. Automatically generated scenario fulfilling the stated properties

fulfilling all constraints. It is partly shown with the white objects and grey associations in Fig. 7. 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.

The former property can be checked in the context of one point in time (snapshot) because it does not depend on dynamic activities. The following example considers a whole scenario. The starting point is again the policy shown in Fig. 7 with small changes. Instead of setting 'wrtUserAssignment' to true we set 'wrtActiveRoles' to true, thus activating the related OCL invariant shown in Sect. 3.2.1, i.e., both roles must not be activated in the same session. Additionally, we forbid the creation of role hierarchy links. Now we would like to check the same property as before. Does a user have the rights to execute both actions? The model validator returns the object diagram shown in Fig. 8. It is similar to the exemplary diagram in Fig. 4. We see that a user can access the resource via both actions within two different successive sessions, as there is no pred/succ link between the objects 'session1' and 'session2'. (The numbers within the generated object names do not have further meaning, i.e., they do not indicate an order within a scenario. The order is determined by the role names 'pred' and 'succ'.) The authorization constraint prohibiting the simultaneous activation of both roles does not apply.

This result shows the policy developer that it may be not sufficient to set just one of the attributes 'wrtUserAssignment' and 'wrtActiveRoles' to true in order to prevent a user from executing both actions ('prepare' and 'approve') on the considered resource. Both corresponding authorization constraints activated in the context of the roles 'clerk' and 'supervisor' still allow the violation of the SoD in specific situations. Additionally, we discovered that both roles must not be related through a role hierarchy. The model validator confirms this assumption, as it does not find a solution which allows a user to execute both actions on the resource while fulfilling all authorization constraints. For this

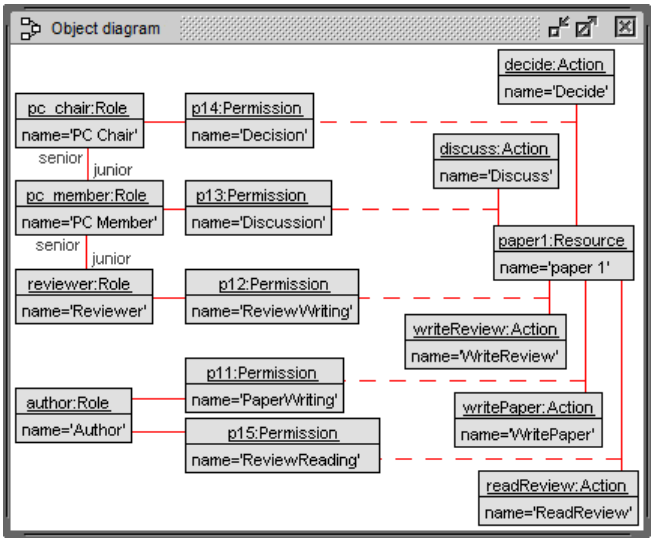

Fig. 9. Basic EasyChair policy without activated authorization constraints

search task we instructed the model validator to check all object diagrams which have up to 30 user, snapshot, session and access objects. Consequently, there is no scenario consisting of 1 to 30 snapshots in which the unwanted activity can be performed. These results are based on the aforementioned precondition that users do not change their roles within a scenario. If we allowed changes, e.g., a role switch from 'clerk' to 'supervisor' in two successive snapshots, a user would still have the rights to execute both actions.

Beside the automatically generated object diagrams, it is also often very 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.

5. Case Study: The EasyChair Conference System

Within this section, we apply our domain-specific language to the EasyChair conference system (Voronkov, 2011), which is well suited for role-based access control and for a discussion of dynamic authorization constraints. We present the configuration of a basic EasyChair policy based on the RBAC metamodel described in Sect. 3 including the activation of different constraints. We also discuss possible extensions to the metamodel with respect to specific authorization demands. The considered authorization constraints are accompanied by example snapshots which demonstrate their effects as well as a detailed explanation of the underlying OCL invariants.

5.1. *Structure of Basic EasyChair Policies*

In our case study, we consider a basic subset of the EasyChair elements and constraints. The first step to defining EasyChair policies is the specification of a corresponding policy structure consisting of roles, resources, actions, permissions and their relationships. This structure represents the basis of concrete EasyChair policies which in addition yield specific configurations of activated authorization constraints. Our basic EasyChair structure should have the following general properties.

— Reviewers are permitted to write paper reviews.
— PC members are allowed to discuss and review papers.
— PC chairs have the right to decide the acceptance or rejection of the available papers. They have also the permissions related to PC members.
— Authors may write papers and read the reviews of their own papers.

Naturally, PC members possess also the right to read paper reviews. In order to keep the presentation of the basic policy in this paper clear, we omit further actions and permissions.

There are several different ways for realizing the described EasyChair structure. In Fig. 9, we show a possible implementation which is based on two main design decisions we made. On the one hand, we chose that a concrete policy should concern one EasyChair event (e. g., a conference or workshop). On the other hand, we chose to exclusively consider papers as resources. Reviews are implicitly covered through respective actions (readReview and writeReview) which directly relate them to the corresponding papers. When a person reviews a paper, for example, the action writeReview is performed on the resource which represents this paper. There is no explicit review resource. However, more complex policies are conceivable, e. g., for handling related events like conferences and special journal issues, or for explicitly modeling further types of resources with individual actions and possible relationships. Those policies may imply extensions to the DSL, i. e., the RBAC metamodel.

Each structural policy element is represented by a UML object, i. e., an instance of the respective metamodel class. The element relationships are determined by links between these objects. Each policy element has further properties which can be configured through the assignment of attribute values. At this point, no constraints are activated, thus all respective attributes are undefined. The presented object diagram focuses the relevant information by hiding undefined attributes.

Within our example structure, the roles ‘PC Chair’, ‘PC Member’ and ‘Reviewer’ are linked through the role hierarchy association. As a consequence, a reviewer has the permission p12 (allowing the review of paper 1), a PC member has the permissions p12 and p13 (allowing the discussion of paper 1), and a PC chair has the permissions p12, p13 and p14 (allowing the decision of paper 1). Instead of utilizing the role hierarchy feature, an analogous condition could be expressed through dependencies between the three roles using the PrerequisiteRoles association, so that, for example, a user who takes the role PC Chair must already hold the roles PC Member and Reviewer. The object diagram further shows an Author role which has the permission to write paper 1 (p11) and to read the reviews of paper 1 (p15).

Although a conference normally involves many submitted papers, we show only one paper in the object diagram. This paper is a representative (template) of further paper instances. The roles, permissions and actions related to paper 1 can be analogously specified for all other papers, thus providing the opportunity to automatically add papers. We define three actions for deciding, discussing and writing a paper as well as two actions for reading and writing the reviews related to a paper. While the actions are independent from the individual papers, the permissions are paper-specific. As a consequence, new papers do not imply new actions, but new permissions. Each further paper x implies analogous permissions (px1 to px5) which are also related to the respective roles. A conference with 30 submissions, for example, is represented by a policy with 30 review writing permissions which are all linked to the role Reviewer. Hence, each user taking this role is generally permitted to review any paper, if no further constraints restrict specific constellations.

We do not involve concrete users in the object diagram, since they do not add information to the structural description of the EasyChair policy. Like concrete papers, users and user assignments can be added later, e. g., manually with respect to the PC members, or automatically during the paper submissions. However, we will make use of example users for illustrating and validating the constraints described in the following sections.

5.2. *Configuring Concrete EasyChair Policies*

As mentioned before, the defined policy structure of EasyChair leads to concrete policies, once the needed authorization constraints have been configured. While conferences, for instance, usually forbid PC chairs to submit papers to the same conference, workshops are often less constrictive in this respect. The EasyChair policies can therefore be configured as necessary. In the following, we consider four concrete security requirements with respect to an example conference.

1 There is only one PC chair.
2 The PC chair is not allowed to be an author in the same conference.
3 Authors must not be allowed to review their own paper. (This requirement is time-independent, e. g., it also covers the possibility that a reviewer is later added as an author of the reviewed paper, as allowed in EasyChair.)
4 Authors are not allowed to read the reviews of their paper, until the acceptance or rejection of their paper has been decided.

The first constraint is activated by assigning the integer value 1 to the attribute maxMembers of the role PC Chair. Figure 10 displays the adapted PC Chair object as well as a situation which violates the activated constraint. The USE tool allows us to focus parts of the policy by hiding the objects and links which are not interesting with respect to a particular question (Gogolla et al., 2011). The hidden elements, however, still exist and can be displayed as needed. This way, even large policies can be comfortably handled. The evaluation of our policy including two PC chairs shows exactly one violated authorization constraint which is named MaximumNumberOfMembers. The underlying OCL invariant has been explained in Sect. 3.2.1.

In order to prevent PC chairs from being authors, we mutually exclude the respective roles by inserting a MutuallyExclusive link between them (see Fig. 11). An object of the association class is related to the new link. The available attributes allow the administrators to determine in which regard the connected roles are exclusive. In our case, the exclusion with respect to the user assignment is appropriate. Thus, we set the attribute wrtUserAssignment to true. A user who takes both roles violates the activated constraint NoUserAssignedToExclusiveRoles.

The respective OCL invariant, which is provided by the RBAC metamodel and is hidden from the administrators of EasyChair events, checks a formula in the context of User objects and their roles, which are considered pairwise. A user-defined OCL query operation userAssignmentExclusives calculates all roles which are marked to be mutually exclusive to the considered role (with respect to the user assignment). Consequently, the invariant states that no role is included in the set of mutually exclusive roles of any other user role.

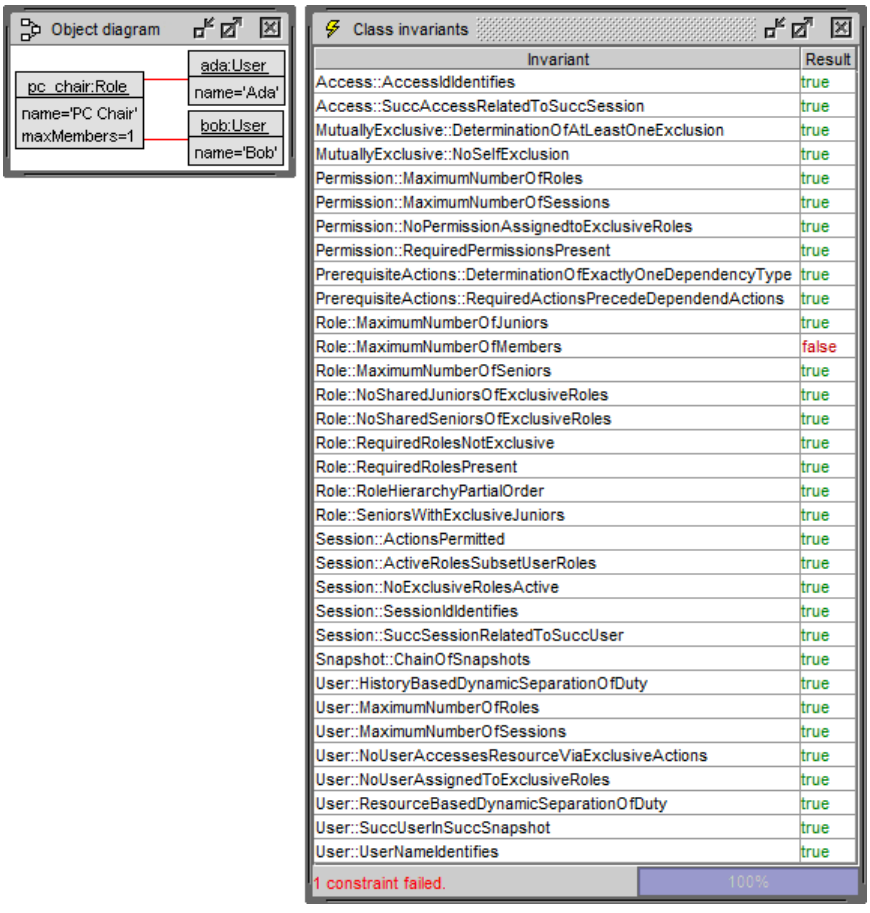

Fig. 10. Situation violating the constraint which limits the maximum number of PC chairs

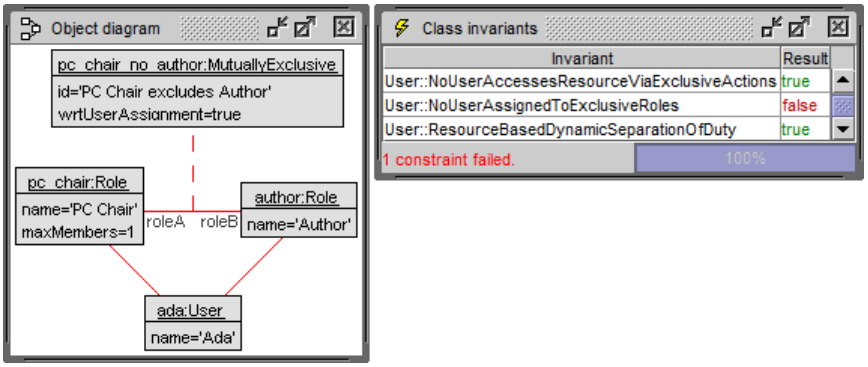

Fig. 11. A PC chair is not allowed to be an author

```
context u:User inv NoUserAssignedToExclusiveRoles:
  u.role->forAll(r1, r2 | r1.userAssignmentExclusives()->excludes(r2))
```

The implementation of the user-defined operation reveals that the association end names roleA and roleB do not have a distinctive meaning, i. e., the semantics does not change when the connected roles switch the association ends. Furthermore, a Role object can simultaneously occur at both association ends, if several exclusions are declared. Consequently, we have to navigate into both directions if we aim to collect all exclusive roles. The expression `mutuallyExclusive[roleA]` (a shorthand version of `self.mutuallyExclusive[roleA]`) describes a navigation from the calling Role object (self) at the association end roleA to the association class objects. In the case of the concrete role PC Chair, we would obtain the object 'pc_chair_no_author'. In the case of the author, we would obtain an empty set, as the Author role occurs at the roleB end. In policies with many exclusions, there may be several respective association class objects linked to a Role object. Since the association class objects may determine different kinds of exclusion, we further have to pick the objects whose attribute wrtUserAssignment holds the value 'true'. By navigating from these selected objects, we reach just the roles which are exclusive with respect to the user assignment.

```
Role::userAssignmentExclusives() : Set(Role) =
  mutuallyExclusive[roleA]->select(wrtUserAssignment).roleB->union(
  mutuallyExclusive[roleB]->select(wrtUserAssignment).roleA)->asSet()
```

The third and fourth named authorization constraints require an extension of the RBAC metamodel. In the following section, we explain the individual extensions with respect to the changes in the DSL and the newly defined UML structures and OCL constraints.

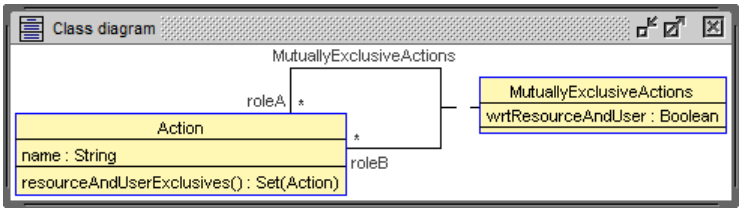

Fig. 12. Extension of the RBAC metamodel in terms of exclusive actions

5.3. *Extending the RBAC Metamodel*

The presented RBAC metamodel comprises the basic concepts and well-known constraints of RBAC. Within this section, we aim to illustrate its extensibility. In many cases, the extension of the DSL consists of (1) adding new UML elements to the metamodel and (2) adding respective OCL invariants. Existing elements can normally remain unchanged. These kinds of extensions are usually performed by UML and OCL experts, while the policy administrators only see the new functionality in the form of the new UML elements available in the RBAC metamodel.

First, we consider the authorization constraint forbidding authors to review their own paper. This constraint cannot be adequately modeled using the existing DSL features. For example, the exclusion of the roles Reviewer and Author with respect to the user assignment would generally prevent reviewers from submitting a paper to the conference. Our aim is, however, to individually manage the permissions with respect to the access of resources.

One way to effectively model the considered constraint is shown in Fig. 12 focusing the class Action and the newly added association class MutuallyExclusiveActions. The association class allows us to mark two actions as mutually exclusive. Analogously to the role exclusion, the boolean attribute determines in which respect the exclusion should hold. In order to prevent users from writing and reviewing the same paper, we set a resource and user as a context (wrtResourceAndUser), i. e., in this regard, a user is not allowed to access a resource via the two respective actions. In other contexts, further attributes may be needed, e. g., an attribute named wrtResource (i. e., two exclusive actions may not be executed on a resource independent from the accessing user, but a user may still execute both actions on different resources), or an attribute wrtUser (i. e., a user may not execute both exclusive actions, but both actions may be executed on the same resource by different users).

Regardless of which attribute is defined, the functionality of the attributes needs to be implemented by a UML and OCL expert. This is a one-time task. In our case, the functionality is realized with the following invariant. After its implementation, the invariant code remains hidden from the EasyChair event administrators who can activate or deactivate the new functionality with respect to specific exclusive actions by switching the boolean value of the new attribute (wrtResourceAndUser).

```
context u:User inv NoUserAccessesResourceViaExclusiveActions:
  Resource.allInstances()->forAll(r |
```

```
    r.access->select(a |
      u.successors()->including(u)->includes(a.session.user)).action->
        forAll(act1, act2|
          act1.resourceAndUserExclusives()->excludes(act2)))
```

This invariant considers each user (u) and each resource (r) existing in a given scenario. First, all actions are collected which are executed by the considered user on the considered resource. This is achieved by navigating from the resource via the accesses to the actions. The accesses are filtered with respect to the users performing them. Only the accesses of the current user are selected. The expression `u.successors()` calculates all User objects representing the considered person over time (i. e., in all successive snapshots). After that, the constraint requires the set of actions the user executes on the resource to not contain mutually exclusive pairs. The invoked OCL query operation resourceAndUserExclusive is defined analogously to the operation userAssignmentExclusives explained before.

The operation 'successors' makes use of an auxiliary operation successorsAux which recursively calculates the transitive closure of the User objects succeeding the calling User object in the following snapshots.

```
User::successors() : Set(User) =
  if self.succ.isDefined then
    successorsAux(Sequenceself.succ)->asSet() else Set endif
User::successorsAux(users:Sequence(User)) : Sequence(User) =
  let successor = users->last().succ in
  if successor.isDefined() then successorsAux(users->append(successor))
  else users endif
```

Figure 13 displays an example scenario violating the introduced authorization constraint. The scenario consists of three successive snapshots and one user (Ada) who at first reviews a paper, then does nothing, and then is added as an author to the same paper. We hide the insignificant parts of the policy and the user action. Since both executed actions are marked as mutually exclusive, this activity is not allowed. Furthermore, Ada would be neither allowed to review a paper after she has been indicated to be an author, nor is she allowed to access a resource via both actions at the same time. The underlying OCL invariant explained before covers all three situations.

The fourth constraint (authors are not allowed to read the reviews of their paper, until the paper was decided) also implies an extension based on the Action class. Figure 14 shows the respective part of the metamodel. Like prerequisite roles, dependent actions can be modeled via a binary reflexive association. However, a plain association would not be sufficient to represent our specific authorization constraint, since it comprises two independent requirements. On the one hand, the action 'ReadReview' requires a paper decision. On the other hand, it requires the user who aims to read the reviews to be an author of the paper. These two requirements differ with respect to the possible users who execute the required action. While the constraint does not determine the user who decides the paper (any PC chair may do this), it requires the same user who aims to read the

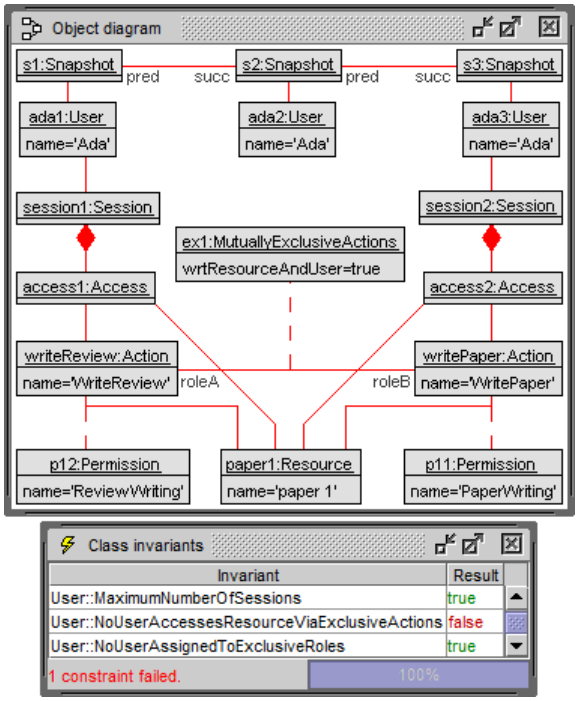

Fig. 13. Scenario violating the paper writing and reviewing restriction

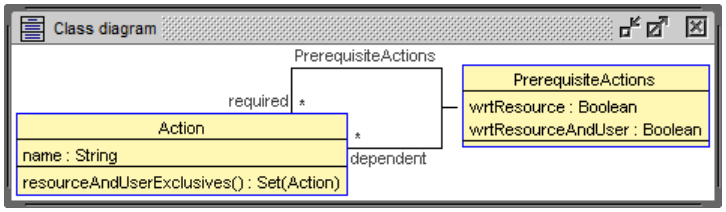

Fig. 14. Extension of the metamodel supporting dependent actions

reviews to be an author of the paper (i. e., to have executed the action 'WritePaper' on the respective paper). Consequently, we added two attributes for specifying this kind of action dependency. If the value of the attribute wrtResource is set to true, the dependent action may be executed on a resource provided that somebody executed the required action on the same resource before (or at the same time). If the attribute wrtResourceAndUser is true, the dependent action may be executed by a user provided that the same user executed the required action on the same resource before.

Figure 15 illustrates the new functionality. In the displayed scenario, Ada is the PC chair who decided the acceptance of paper 1 in the first snapshot. At the same time, Bob has been marked as an author of paper 1. This situation allows Bob to read the reviews of paper 1 (second snapshot). The validity of this scenario can be corrupted in different ways. If we remove 'access1' or 'access2', the new authorization constraint is violated

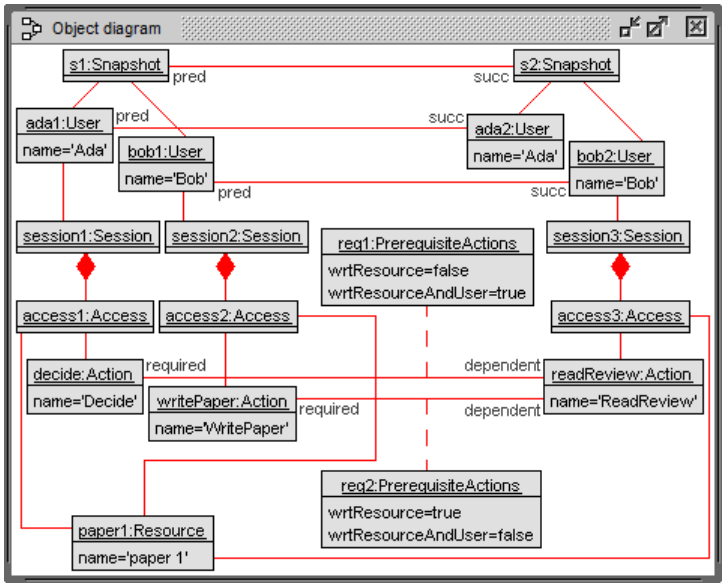

Fig. 15. Scenario showing a valid execution of dependent actions

because the paper decision and paper writing is required. Furthermore, if we switched the attribute values of the object 'req2' which determines the dependency between the paper decision and the reading of the paper reviews (by setting wrtResource to 'false' and wrtResourceAndUser to 'true'), the constraint would be violated because Ada decided the paper instead of Bob.

The underlying OCL invariant which accompanies the new authorization constraint is shown below. It checks the respective property for each object (m) of the association class PrerequisiteActions, each resource (r) and each access (acc) which is related to the dependent action and the considered resource. The Access object is indirectly linked to a specific User object which is in turn linked to a snapshot. The constraint demands that the required action is executed in the context of this snapshot or its predecessors (i. e., in the present or the past). If the attribute wrtResource is true, the constraint considers all possible accesses with respect to the required action. Otherwise (wrtResourceAndUser is true) only the accesses of the user who belongs to the access with respect to the dependent action are considered.

```
context m:PrerequisiteActions inv RequiredActionsPrecedeDependentActions:
  Resource.allInstances()->forAll(r |
    m.dependent.access->select(resource = r)->forAll(acc |
      let u = acc.session.user in
      u.snapshot.predecessors()->including(u.snapshot)->exists(snap |
        let relevantAcc =
          if wrtResource then snap.user.session.access
```

```
        else snap.user->select(name = u.name).session.access endif in
    relevantAcc->select(resource = r).action->includes(m.required))))
```

Within the invariant we make the assumption that exactly one of the two attributes is set to true. We ensure this property with a further invariant.

```
context m:PrerequisiteActions inv DeterminationOfExactlyOneDependencyType:
  wrtResource xor wrtResourceAndUser
```

5.4. *Evaluation of Concrete Scenarios*

We applied the UML-based Specification Environment for creating and analyzing the RBAC metamodel as well as for developing and analyzing the RBAC policies explained in previous sections. In this section, we describe how the USE system can be applied to evaluate real RBAC scenarios, in order to check their conformance to the underlying policy.

In the context of the EasyChair system, a realtime validation is probably not necessary, since the users do not continuously perform accesses to changing resources. Thus, the validation of an event scenario may be invoked manually as needed, e. g., for checking the event after its initialization, after the submission deadline and the reviewer assignment, or after the paper decisions. If an EasyChair policy has been developed and analyzed in USE, it is advantageous to apply USE again for validating real scenarios. However, these scenarios normally comprise many users and submitted papers. Object diagrams including all elements of a real event become very large and are thus difficult to analyze. The USE system allows the policy administrators to inspect large policies through different views and mechanisms which are explained below.

As an example, we created an event with about 50 users, 30 papers and various user activities. The first step for checking the validity of this scenario is the class invariants view displayed in Fig. 16 which shows the violation of one authorization constraint. Since the constraint is checked in the context of User objects, one or more users must provoke the violation. The USE class extend view lists the objects of a selected class. It further relates the evaluation result of each constraint defined in the context of the selected class to the individual objects. Figure 16 also reveals the activities of user Ike to be unauthorized, i. e., he accesses resources with exclusive actions.

Instead of searching the object diagram to inspect the forbidden resource access, we textually query the scenario with a simple OCL expression. After navigating from the object ike3 and its successors to the performed accesses, we collect all actions (a) executed on the resources (r) in the form of a tuple. The evaluation of the expression is shown in Fig.16. We see that Ike has written and reviewed the same paper (number 14).

6. Related Work

Dynamic and specifically history-based access control policies have long been discussed in literature. The Clark-Wilson model first discussed forms of history-based SoD for banking

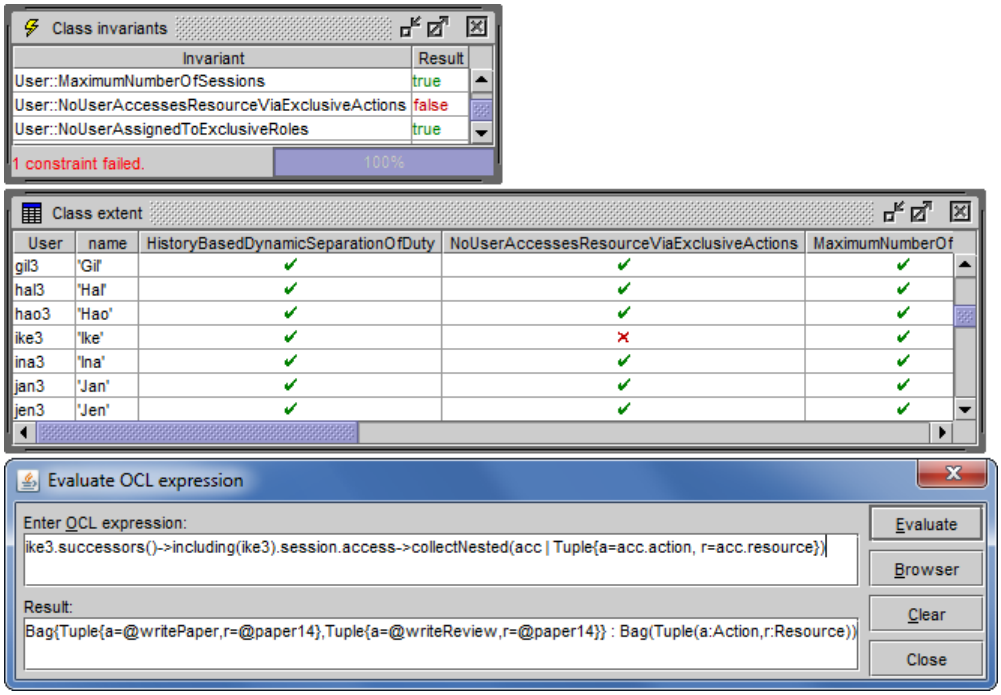

Fig. 16. Validating large EasyChair scenarios in USE

transactions (Clark and Wilson, 1987). In addition, Sandhu's Transaction Control Expressions (TCE) allowed one to specify the access history on resources (objects) (Sandhu, 1988). Later, Simon and Zurko defined a taxonomy for SoD and in this context introduced resource-based dynamic SoD and history-based dynamic SoD as flexible variants of SoD (Simon and Zurko, 1997). Gligor et al. formalized these definitions in first-oder LTL, but came up with complex formulae, explicitly talking about states (Gligor et al., 1998). Schaad formalized RBAC and variants of history-based SoD with the Alloy language and carried out validation with the Alloy analyzer (Schaad, 2003). Again, the formulae became complex because the state had to be explicitly represented.

Qunoo and Ryan define a logic-based policy language called X-Policy in order to express dynamic access control policies (Qunoo and Ryan, 2010). They also use the EasyChair system as a case study to demonstrate the expressiveness of their language. However, tool support is not available yet.

There are also approaches, which aim to specify stateful access control policies, such as history-based SoD, in linear temporal logic (Barth et al., 2006; Dougherty et al., 2006; Mossakowski et al., 2003). This way, temporal operators can be used to express time dependencies in policies rather than explicitly talking about states. Some work has been carried out regarding the validation of dynamic policies, e.g., with model checking (Schaad et al., 2006; Zhang et al., 2008).

All described approaches, however, employ logic-based formalisms and related tools, which are difficult to use if one is not an expert in formal methods. In contrast to

our approach, it was not the aim of those works to develop a DSL, which hides the technical details of the specified authorization constraints. In this respect, the SecPAL authorization language is closer to our approach (Becker et al., 2010). SecPAL is also logic-based with the semantics, e. g., given by a translation to Datalog. Due to the fact that SecPAL's syntax is closer to natural language it can be seen as a DSL. Our approach, however, is based on UML, a mainstream modeling language, with a rich tool support. In particular, our DSL employs only class and object diagrams, which are widely used in industry. This way, a security officer does not have to learn a new formalism and hence the barrier to using the DSL is lower.

There is a plethora of works integrating security policies into system models based on UML such as (Sohr et al., 2008a; Jürjens, 2002; Ray et al., 2004; Ahn and Shin, 2001; Fernández-Medina and Piattini, 2004; Basin et al., 2006). Some of the approaches do not particularly address RBAC like UMLsec (Jürjens, 2002). Basin et al. (Basin et al., 2006) present the modeling language SecureUML for integrating the specification of access control into application models and for automatically generating access control infrastructures for applications. They also deal with authorization constraints, but do not support SoD constraints. In (Sohr et al., 2008a), 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. (Ray et al., 2004) 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.

Several works on the validation of RBAC policies based on UML and OCL have been presented (Basin et al., 2009; Yu et al., 2008; Sohr et al., 2008a; Höhn and Jürjens, 2003). 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?' (Basin et al., 2009). Although not explicitly addressed in this paper, our approach allows the same kind of queries through the query facility of the USE tool (Gogolla et al., 2007) into which the model validator is integrated. In (Yu et al., 2008), 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.

Our SAT-based model validator approach developed and applied for comprehensive validation of UML and OCL models is related to UML2Alloy (Anastasakis et al., 2007), a method for translating UML and OCL into Alloy specifications, since the Alloy language is implemented in the 'Alloy Analyzer' whose current version is based upon Kodkod. However, UML2Alloy does not yet support some frequently used UML and OCL features like n-ary associations, association classes or standard operations on integer values which are provided by the model validator.

7. Conclusion

We presented an RBAC metamodel as a basis for an RBAC DSL allowing security officers to design complex policies and to analyze explicit and implicit properties without handling the often very complex underlying textual constraints. The properties can be time-dependent (dynamic) and time-independent (static) corresponding to the nature of the authorization constraints which may relate past, present and future activities at the end-user level. We discussed the need for analysis and validation. Even small changes to a model can imply new implicit properties that the developer may not think of, regardless of whether we consider the design of a policy or the development of the RBAC metamodel itself.

The current RBAC metamodel comprises basic concepts and authorization constraints. It is designed as a groundwork for versatile extensions like advanced role delegation and revocation concepts. We plan to extend the model in different ways in order to achieve a mature RBAC framework including all concepts for practical application. The approach must be improved by performing larger and more practically relevant case studies in the context of real industrial environments. Another direction of our work concerns polishing the user interface. Currently, many tasks require UML and OCL know-how. An improved user interface may hide unneeded details.

References

Abi Haidar, D., Cuppens-Boulahia, N., Cuppens, F., and Debar, H. (2006). An extended RBAC profile of XACML. In *Proceedings of the 3rd ACM workshop on Secure web services*, SWS '06, pages 13–22, New York, NY, USA. ACM.

Ahn, G.-J. and Shin, M. E. (2001). Role-Based Authorization Constraints Specification Using Object Constraint Language. In *Proc. of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 157–162. IEEE.

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

Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2007). UML2Alloy: A Challenging Model Transformation. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007*, volume 4735 of *LNCS*, pages 436–450. Springer, Berlin.

Barth, A., Datta, A., Mitchell, J. C., and Nissenbaum, H. (2006). Privacy and Contextual Integrity: Framework and Applications. In *IEEE Symposium on Security and Privacy*, pages 184–198. IEEE Computer Society.

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

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

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

Bertino, E., Ferrari, E., and Atluri, V. (1999). The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104.

Clark, D. C. and Wilson, D. R. (1987). A comparison of commercial and military security policies. In *Proc. IEEE Symp.on Security and Privacy, Washington DC*.

Dougherty, D. J., Fisler, K., and Krishnamurthi, S. (2006). Specifying and Reasoning About Dynamic Access-Control Policies. In Furbach, U. and Shankar, N., editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer.

Fernández-Medina, E. and Piattini, M. (2004). Extending OCL for secure database development. In *Proc. of UML 2004 - The Unified Modeling Language: Modeling Languages and Applications*, volume 3273 of *LNCS*, pages 380–394. Springer.

Gligor, V. D., Gavrila, S. I., and Ferraiolo, D. (1998). On the formal definition of separation-of-duty policies and their composition. In *1998 IEEE Symposium on Security and Privacy (SSP '98)*, pages 172–185. IEEE.

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

Gogolla, M., Hamann, L., Xu, J., and Zhang, J. (2011). Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In *Proc. 10th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'2011)*.

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

Hilty, M., Pretschner, A., Basin, D. A., Schaefer, C., and Walter, T. (2007). A Policy Language for Distributed Usage Control. In Biskup, J. and Lopez, J., editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546. Springer.

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

Jürjens, J. (2002). UMLsec: Extending UML for secure systems development. In Jézéquel, J.-M., Hussmann, H., and Cook, S., editors, *Proceedings of The Unified Modeling Language - Model Engineering, Concepts, and Tools, UML 2002*, volume 2460 of *LNCS*, pages 412–425. Springer, Berlin.

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

Kuhlmann, M., Hamann, L., and Gogolla, M. (2011a). Extensive Validation of OCL Models by Integrating SAT Solving into USE. In Bishop, J. and Vallecillo, A., editors, *Proc. of the 49th International Conference on Objects, Models, Components and Patterns, TOOLS 2011*, LNCS. Springer, Berlin.

Kuhlmann, M., Sohr, K., and Gogolla, M. (2010). RBAC Metamodel: Sources and Validation Results. `http://www.db.informatik. uni-bremen.de/publications/Kuhlmann_2010_RBAC_sources.pdf`.

Kuhlmann, M., Sohr, K., and Gogolla, M. (2011b). Comprehensive Two-level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In Baik, J., Massacci, F., and Zulkernine, M., editors, *Fifth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011*. IEEE Computer Society.

Li, N., Tripunitara, M. V., and Bizri, Z. (2007). On mutually exclusive roles and separation-of-duty. *ACM Trans. Inf. Syst. Secur.*, 10.

Mossakowski, T., Drouineaud, M., and Sohr, K. (2003). A temporal-logic extension of role-based access control covering dynamic separation of duties. In *Proc. of TIME-ICTL 2003, Cairns, Queensland, Australia*.

Nash, M. J. and Poland, K. R. (1990). Some conundrums concerning separation of duty. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 201–207.

Object Management Group (2010a). Object Constraint Language - Version 2.2. `formal/2010-02-01`.

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

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

Qunoo, H. and Ryan, M. (2010). Modelling dynamic access control policies for web-based collaborative systems. In *Proceedings of the 24th annual IFIP WG 11.3 working conference on Data and applications security and privacy*, DBSec'10, pages 295–302, Berlin, Heidelberg. Springer-Verlag.

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

Richters, M. and Gogolla, M. (2001). OCL - Syntax, Semantics and Tools. In Clark, T. and Warmer, J., editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley Professional, Boston, Massachusetts, second edition.

Sandhu, R. (1988). Transaction control expressions for separation of duties. In *Proc. of the Fourth Computer Security Applications Conference*, pages 282–286.

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

Schaad, A. (2003). *A Framework for Organisational Control Principles*. PhD thesis, University of York,UK.

Schaad, A., Lotz, V., and Sohr, K. (2006). A model-checking approach to analysing organisational controls in a loan origination process. In *Proc. of the 11th ACM Symposium on Access Control Models and Technologies*, New York. ACM Press.

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

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

Sohr, K., Mustafa, T., Bao, X., and Ahn, G.-J. (2008b). Enforcing Role-Based Access Control Policies in Web Services with UML and OCL. In *Proceedings of the 23th Annual Computer Security Applications Conference*, pages 257–266. IEEE Computer Society.

Torlak, E. and Jackson, D. (2007). Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems - 13th International Conference, TACAS 2007*, volume 4424 of *LNCS*, pages 632–647. Springer, Berlin.

Voronkov, A. (2011). EasyChair conference system. `http://www.easychair.org/`.

Yu, L., France, R. B., and Ray, I. (2008). Scenario-Based Static Analysis of UML Class Models. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008*, volume 5301 of *LNCS*, pages 234–248. Springer, Berlin.

Zhang, N., Ryan, M., and Guelev, D. P. (2008). Synthesising Verified Access Control Systems through Model Checking. *Journal of Computer Security*, 16(1):1–61.

Zhang, X., Parisi-Presicce, F., Sandhu, R., and Park, J. (2005). Formal model and policy specification of usage control. *ACM Transactions on Information and System Security*, 8(4):351–387.