Analyzing and Managing Role-Based Access Control Policies

Karsten Sohr, Michael Drouineaud, Gail-Joon Ahn, and Martin Gogolla

Abstract

Today more and more security-relevant data is stored on computer systems; security-critical business processes are mapped to their digital counterparts. This situation applies to various domains such as health care industry, digital government, and financial service institutes requiring that different security requirements must be fulfilled. Authorization constraints can help the policy architect design and express higher-level organizational rules. Although the importance of authorization constraints has been addressed in the literature, there does not exist a systematic way to verify and validate authorization constraints. In this paper, we specify both non-temporal and history-based authorization constraints in the Object Constraint Language (OCL) and first-order linear temporal logic (LTL). Based upon these specifications, we attempt to formally verify role-based access control policies with the help of a theorem prover and to validate policies with the USE system, a validation tool for OCL constraints. We also describe an authorization engine, which supports the enforcement of authorization constraints.

Index Terms

Role-based access control policy, authorization constraints, linear temporal logic, Object Constraint Language.

I. Introduction

Information technology pervades more and more our daily life. This applies to different domains such as health care, e-government, and banking. New technologies, however, go along with new risks, which must be systematically dealt with. Specifically, the information contained in the IT systems can be regarded as a key resource of an organization and must therefore be protected adequately. Due to the fact that insider attacks are a major threat for large organizations [1] it is mandatory to establish adequate mechanisms that enforce the access control requirements demanded by the rules and laws relevant to an organization.

For example, in Europe strong privacy requirements such as those formulated in the Directive 95/46/EC [2] exist. This directive among other areas applies to hospitals, and hence specific organizational rules must be implemented in order to prevent privacy violations. A typical organizational rule in a hospital might be "a nurse can only see the records of all patients who have been on her ward within the previous 90 days". Another rule might state that a physician can only delegate the permission to read a patient record to another physician who has at the same time activated a specialist role.

Dr. Sohr is with the Universität Bremen, Germany.
Mr. Drouineaud is with the Universität Bremen, Germany.
Prof. Dr. Gail-Joon Ahn is with the University of North Carolina at Charlotte, USA.
Prof. Dr. Gogolla is with the Universität Bremen, Germany.

In the banking domain, other protection goals such as data integrity and accountability are more important. In particular, separation of duty policies (SoD) [3], [4] must be enforced. SoD is a well-known principle that prevents fraud and error by requiring that at least two persons are required to complete a task. A common example of such an SoD rule is “a clerk must not prepare and approve a check”.

As pointed out by Sandhu et al. [5], one of the main advantages of role-based access control (RBAC) is that such higher-level organizational rules can be implemented in a natural way. Specifically, advanced RBAC concepts like role-based authorization constraints [1] are an important means for laying out higher-level organizational rules [6]. Due to the fact that RBAC has also become an accepted standard for access control, we consider RBAC as the basic access control model within this paper.

Usually, the control and protection goals of an organization can only be expressed by a set of rules rather than a single rule. Hence, we define an RBAC policy as hierarchical RBAC in the sense of the RBAC standard [7] plus the *set of authorization constraints* defined for the organization in question. Unfortunately, authorization constraints can increase the complexity of RBAC itself. Thus, it generally becomes more difficult to make sure that certain required properties hold in a given RBAC policy or that the policy is free of contradictions. Through the combination of different types of authorization constraints undesirable properties may arise. For example, assume that we have defined the SoD rule mentioned above. Let us further assume that a specific user has the permission for the approval of the check and that she can delegate this permission to other users. If the delegatee already has the permission to prepare the check in question, then either the SoD property is violated or the delegation cannot be executed.

New technologies such as Web services increase the complexity and the dependencies of IT systems w.r.t. access control. For this reason, it is an important task to find a suitable methodology to express and verify assertions on organization-wide RBAC policies. Otherwise, unauthorized access and consequently fraud may be expected. Although there are several works on the specification of authorization constraints, e.g., [6], [8], there is a lack of an appropriate tool support for the analysis/verification of RBAC policies.

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

The work presented in this paper can be regarded as a first step to help security officers in the definition process of consistent (i.e, free of contradictions) and correct RBAC policies. In addition, even if we have defined correct RBAC policies, we also wish to enforce these policies automatically by the IT system. Hence, tool support for the enforcement of RBAC policies is also desirable.

In the following, we briefly discuss two different approaches to the policy specification and verification. The first approach is more formal and uses a theorem prover [9] for the verification. In contrast, the second one is more practical and is based upon a validation tool for software models [10]. In any case, the verification should be carried out in the design process of the policy and not after the deployment. This way, only consistent RBAC policies are implemented, and policies with undesirable properties are ruled out early. Interestingly, the validation tool can also be employed to implement an authorization engine, which helps to *enforce* certain RBAC policies. This topic is discussed after the verification sections.

A. Formal Specification and Verification

In domains with high security requirements such as banking applications, which deal with large amounts of money, or military applications, a rigorous way for the specification and analysis of RBAC policies is needed. Specifically, this is required for the certification of an IT system w.r.t. the Common Criteria for high levels of security [11].

We consider first-order linear temporal logic (LTL) [12] an appropriate formalism for specifying RBAC policies because often one must cope with dynamic policies. In banking applications, for example, dynamic SoD authorization constraints must be enforced such as history-based SoD [13]. History-based SoD is a flexible variant of SoD, in which a user may have all the privilege for a business process, but must not perform all the subtasks of this process on a certain object (e.g., check). Furthermore, we have often other types of dynamic authorization constraints such as delegation rules or authorization constraints which mandate a certain order of task execution as needed for workflows. In these cases, first-order LTL allows for a concise and elegant formulation of the RBAC policy in question. Furthermore, first-order LTL has been extensively studied in the literature [14], [12].

The formal verification assures that an RBAC policy satisfies the properties intended by the organization (e.g., no user may prepare and approve a check). We will carry out this verification by means of the theorem prover Isabelle [9]. In particular, the first-order LTL (including axioms) is embedded into Isabelle. Afterwards, the proofs of those properties to be fulfilled by an RBAC policy are automatically checked by Isabelle. The details of this formal verification are presented later in this paper.

B. Practical Specification and Validation

Due to the fact that theorem provers are still tools for experts, the aforementioned formal verification approach is strictly speaking not necessary in applications with moderate security requirements. For this reason, we present a more light-weight approach to policy specification and analysis. Clearly, we can employ this approach in application domains with high security demands, too, and then use the formal verification if certain problems have been detected in the practical verification step.

For the practical specification and verification, we employ the Unified Modeling Language (UML) and Object Constraint Language (OCL)[2] [15]. As demonstrated in [16], [17], UML/OCL can be conveniently used to specify several types of authorization constraints. Moreover, owing to the fact that OCL has proved its applicability in several industrial applications, OCL is an appropriate means for such a practically relevant process as the design of RBAC policies.

Hence, we demonstrate in this paper how to employ the USE system (UML-based Specification Environment) [10] to verify RBAC policies formulated in UML and OCL. USE is a validation tool for UML models and OCL constraints, which has been reportedly applied in industry and research. We consider validation by generating snapshots (system states) as prototypical instances of a UML/OCL model and compare the generated instances with the specified model (i.e., the RBAC policy in our case). From now on, we use the term "validation" introduced by Richters et al. [10] whenever we discuss the practical verification.

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

Validation with the USE system can help to detect if certain constraints conflict with each other or if constraints are missing. The latter case may lead to a situation in which unallowed access is possible. For example, if an SoD constraint between a *cashier* and a *cashier supervisor* is missing, a user who assumes both roles may commit fraud. The former case, however, may have the effect that from the security point of view reasonable system states cannot be established. While the validation allows the detection of certain conflicting constraints, one cannot *formally prove* the correctness of the RBAC policy in question. For this purpose, a more formal approach is required such as theorem proving.

C. Authorization Engine

Beyond specification and validation/verification, the enforcement of RBAC policies is also an important issue. Such an authorization engine should be designed and implemented by sound software engineering techniques. In particular, the main focus should lie in the modeling process, whereas the implementation should be carried out routinely and as much as possible automatically. In this paper, we also demonstrate how to employ the USE system, which is a validation tool for software models, to implement an authorization engine. This engine serves as a key component for enforcing various types of authorization constraints. This way, RBAC policies for different organizations can be implemented.

D. Structure of this Paper

The remainder of the paper is now organized as follows: Section II gives a short overview of RBAC and authorization constraints. In Section III, typical dynamic authorization constraints such as history-based SoD and delegation rules are formalized in first-order LTL. Thereafter, we present the formal verification of RBAC policies with Isabelle. Section IV demonstrates how RBAC policies can be specified in UML/OCL and validated by means of USE. Section V describes an authorization engine built on the USE system whereas Section VI discusses extensions of our work and related work. Section VII summarizes the results of this paper.

II. RBAC AND AUTHORIZATION CONSTRAINTS

RBAC [5], [7] has received considerable attention as a promising alternative to traditional discretionary and mandatory access control. The explicit representation of roles simplifies the security management and makes possible to use security principles like SoD and least privilege [5]. We now give an overview of (general) hierarchical RBAC according to the RBAC standard [7]:

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

Users may activate a subset of the roles they are assigned to in a *session*. P is the set of ordered pairs of operations and objects. In the context of access control all resources accessible in an IT-system (e.g., files, database tables) are referred to by the notion *object*. An *operation* is an active process applicable to objects (e.g., read, write, append). The relation PA assigns to each role a subset of P. So PA determines for each role the operation(s) it may execute and the object(s) to which the operation in question is applicable for the given role. Thus any user having assumed this role can apply an operation to an object if the corresponding ordered pair is an element of the subset assigned to the role by PA.

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 are authorization constraints. Authorization constraints can be regarded as restrictions on the RBAC functions and relations. For example, an SoD constraint may state that no user may be assigned to both the *cashier* and *cashier supervisor* role, i.e., the UA relation is restricted. It has been argued elsewhere that authorization constraints are the principal motivation behind the introduction of RBAC [5]. 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 such as various types of static and dynamic SoD constraints [13], [18], [6]; constraints on delegation [19], [20];

cardinality constraints [5]; context constraints [19], [21]; workflow constraints [22].

As indicated above, we now define an RBAC policy as follows:

Definition 1: An RBAC policy of an organization is hierarchical RBAC plus the set of authorization constraints defined for the organization in question.

An RBAC policy should be distinguished from an *RBAC configuration*:

Definition 2: An RBAC configuration consists of the concrete roles, permissions, users, sessions, role assignments and role hierarchies currently defined within an organization.

III. Formal Specification and Verification of RBAC Policies

In the following, we describe our formal approach to the specification and verification of RBAC policies in more detail. We begin with the specification of dynamic authorization constraints, namely, dynamic SoD, delegation and revocation rules, in first-order LTL. Later, these authorization constraints are used in an example, which demonstrates how the verification works.

A. Formal Specification of RBAC Policies in LTL

First-order LTL [12] is a linear temporal first-order logic. Thus it is powerful enough to express most of mathematics such as Zermelo-Fraenkel set theory (including the axiom of choice). Therefore it is sufficient for reasoning about RBAC policies on a conceptual level [3]. On the other hand, there is a clearly defined formal semantics based upon Kripke frames for first-order LTL [12]. Since it is a linear temporal logic, it is specifically well-suited for reasoning about temporal invariants as needed for IT security and for safety-critical systems such as aircraft controllers or railway control systems [14].

Subsequently, we explain the basic concepts of first-order LTL. A temporal first-order signature consists of a set of sorts, a set of function symbols and a set of predicate symbols (each symbol coming with a string of argument sorts and, for function symbols, a result sort). There are *rigid* and *flexible* predicate symbols: the former do not change over time, whereas the latter may vary. Models live over discrete time,

[3]Ahn demonstrates that the specification language RCL 2000 for authorization constraints is equivalent to a restricted form of first-order predicate logic [6].

spec TEMPORALRBAC =
  sorts $User, Session, Role, Operation, Object$
  rigid op $user : Session \to User;$
  flexible preds $UA : User \times Role;$
    $PA : Operation \times Object \times Role;$
    $\_\_Active\_in\_\_ : Role \times Session;$
    $Exec : Session \times Operation \times Object;$
    $Exec : User \times Operation \times Object$
  forall $r : Role;\ u : User;\ s : Session;\ op : Operation;\ obj : Object$
  • $(\Diamond r\ Active\_in\ s) \Rightarrow UA(user(s), r)$
  • $\Box(Exec(s, op, obj) \Rightarrow \exists r : Role\ .\ r\ Active\_in\ s \wedge PA(op, obj, r))$
  • $\Box(Exec(u, op, obj) \Leftrightarrow \exists s : Session.user(s) = u \wedge Exec(s, op, obj))$
end
spec DELEGATETEMPORALRBAC=TEMPORALRBAC **then**
  rigid pred $UAO : User \times Role;$
  flexible preds $UA : User \times Role;$
    $UAD : User \times Role;$
    $...\ further\ predicates,\ e.g.,\ Delegate,\ Revoke$
  forall $u : User;\ r : Role$
  • $UAD(u, r) \vee UAO(u, r) \Leftrightarrow UA(u, r)$
  • $\neg(UAD(u, r) \wedge UAO(u, r))$
  • $...\ see\ text$
end

Fig. 1. Temporal-logic RBAC and predicates for delegation, formalized within first-order LTL.

indexed by the natural numbers as time steps. They interpret the sorts with (time-independent) carrier sets, function and rigid predicate symbols with time-independent functions and predicates of appropriate types, and flexible predicate symbols with families of functions and predicates, where the families are indexed by natural numbers.

Sentences are the usual first-order sentences built from equations, predicate applications and logical connectives and quantifiers $\forall$, $\exists$. Additionally, we have the modalities $\Box$ (always in the future), $\Diamond$ (sometimes in the future) and $\bigcirc$ (in the next step). The corresponding past modalities are $\boxminus$, $\diamondminus$ and $\ominus$. Satisfaction is defined inductively for a given time step, where the modalities allow for referring to other time steps. A sentence is satisfied in a model if it is satisfied in the time step zero.

We now specify RBAC in first-order LTL (see Fig. 1, upper part). For this purpose, we use the notation of the algebraic specification language CASL as we have done earlier [23]. The function $user$ is rigid (i.e., does not depend on the state) whereas the predicates UA, PA, $Active\_in$ and $Exec$ are flexible (i.e., depend on the state). $Exec$ now traces the operations performed, i.e., $Exec(s, op, obj)$ means that session s executes operation op on object obj in the present (implicit) state. Note that we have left out the predicates and axioms for role hierarchies to simplify the discussion.

1) Specification of Dynamic SoD with History: Dynamic SoD is a flexible form of SoD. Here, a user may perform certain steps of a task, but only if she has not done certain other steps of the task before. There are several attempts to express dynamic SoD in the computer security world such as Sandhu's Transaction Control Expressions [24]. Practical applications of RBAC often need dynamic SoD; see Simon and Zurko [18] and Nash and Poland [3]. In fact, Sandhu stresses the importance of history [24].

Dynamic SoD properties [13], [18] can be elegantly formulated in first-order LTL without explicitly talking about states. We demonstrate this by means of two typical examples, namely, object-based dynamic SoD and history-based SoD. Note that we use the CASL-style representation again and hence the variables after the **forall** quantifier correspond to the parameters of the SoD predicates. The SoD predicates are defined by equivalences, similarly to the second $Exec$ predicate given in Fig. 1:

- Object-based dynamic SoD

  A user may perform at most one operation on a given object.

  forall $obj : Object;$
  $ObjDSoD(obj) \Leftrightarrow [\forall u : User;\ op, op' : Operation. op \neq op' \wedge Exec(u, op, obj) \Rightarrow \Box\neg Exec(u, op', obj)]$

- History-based dynamic SoD

  A user may execute all operations and may also execute more than one operation on a target object, but she may not perform all operations (if more than one) on the same target object.

  forall $obj : Object;$
  $HDSoD(obj) \Leftrightarrow [\forall u : User;\ op : Operation.$
  $\quad Exec(u, op, obj) \Rightarrow\ \exists op' : Operation.(op \neq op' \wedge \boxminus \neg Exec(u, op', obj)) \vee \forall op' : Operation.op' = op]$

First-order LTL can also be employed in order to formally specify SoD policies where the order of executions matters, e.g., the dynamic object-based SoD variant, proposed by Nash and Poland [3]. The $Exec$ predicate can then be used to express RBAC policies for workflows where the tasks are executed in a certain order [22]. This is, however, discussed elsewhere due to space limitations [25].

2) Delegation and Revocation: In the following, both delegation and revocation policies are formalized within first-order LTL.

Delegation: Delegation is an important factor to fulfill dynamic requirements for secure distributed computing environment. There are many definitions of delegation in the literature [26], [27], [28], [29], [30]. In general, it is referred to as one active entity in a system delegates its authority to another entity to carry out some functions on behalf of the former.

As proposed by Zhang et al. [30], we introduce two new user assignment relations, called original user assignment UAO and delegated user assignment UAD. This way, one can make explicit whether a role is assigned to the user by an administrator directly or a role was delegated by another user. UA then is the union of these two assignment relations as shown in the lower part of Fig. 1.

Subsequently, we formalize a basic variant of delegation in first-order LTL. In this variant, user u delegates role r to user $u1$ and loses at the same time the power of the delegated role r:

$$\textbf{forall } u, u1 : \mathit{User};\ r, r1 : \mathit{Role};$$
$$\mathit{Delegate}(u, r, u1, r1) \Leftrightarrow [UAO(u, r) \wedge UA(u1, r1) \wedge \neg UA(u1, r) \wedge \mathit{const} \Rightarrow \bigcirc(UAD(u1, r) \wedge \neg UA(u, r))]$$

$\neg UA(u1, r)$ is required here because it is not useful to delegate role r to a user $u1$ who has already been assigned to this role. Moreover, u must obviously belong to r on delegation, and often $u1$ should hold the power of a certain prerequisite role $r1$ on the delegation process. Sometimes, there also exist additional constraints (e.g., concerning the delegation depth or time) which are to be satisfied to enable the delegation process, i.e., the delegation depends on further predicates. These delegation constraints are represented by the const statement, which is a first-order LTL formula.

Revocation: Often, it is necessary to revoke roles that have been delegated, e.g., when a clinician returns from vacation. Several different semantics are possible for user revocation such as [31], [26]. We give here only the formal specification of a simple revocation rule:

$$\textbf{forall } u : \mathit{User};\ r : \mathit{Role};\ \ \mathit{Revoke}(u, r) \Leftrightarrow [UAD(u, r) \wedge \mathit{const} \Rightarrow \bigcirc\neg UAD(u, r)]$$

const stands for a revocation constraint. For example, the power to read a patient's electronic health record is lost if the clinician does not belong to the current department of the patient in question any more.

B. Formal Verification of RBAC Policies

Having specified various types of authorization constraints in first-order LTL, we now generate RBAC policies by combining several authorization constraints. But doing so, we want to be sure that our RBAC policies meet the requirements (see end of Section III-B.2), and at the same time avoid unintended side effects like deadlocks (cf. example in Section III-B.2) or unallowed access. For this reason, we treat the formal verification of RBAC policies in this section. Formal verification may be cumbersome, but it guarantees the highest degree of reliability. Moreover, security assurance standards such as the Common Criteria request formal methods as a precondition for the highest security level (EAL7). Here, we will use the theorem prover Isabelle as a tool for the formal verification.

Isabelle [9] is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. Formal calculi and/or mathematical theorems are encoded in theory files (suffix `.thy`) that can be invoked to Isabelle. Invoking a theory file makes Isabelle accept all definitions, axioms, etc. in this theory. Furthermore, Isabelle will check proofs of theorems given in this theory.

1) How to Build Correct RBAC Policies: The first-order LTL with past modalities specified in Isabelle/HOL as `LTL.thy` is a powerful formalism for expressing and examining RBAC policies [32]. The most important properties of this kind of LTL have been proved as theorems in `Basic_Inf_Rules.thy`, `Basic_Op_Rules.thy` and `Adv_Op_Rules.thy` in this sequence, which was a demanding task. RBAC is defined in `RBAC1.thy` [32]. However, this can only help if the specified RBAC policy is consistent. Consistency is to be understood in the following sense:

Definition 3: An RBAC policy is consistent if a sequence of non-empty RBAC configurations exists which satisfies all the rules of the policy in question.

Note that we consider sequences of RBAC configurations here and not only a single RBAC configuration because we use a linear temporal logic as the formalism for policy specification. A sequence of RBAC configurations is then a model of the RBAC policy in first-order LTL, i.e., the RBAC policy is free of contradictions. Finding such a sequence for which a given RBAC policy holds proves the satisfiability of the considered RBAC policy in the sense of first-order LTL. Thus, applying adequate tools such as model checkers or the USE tool can already help the policy designer to avoid basic mistakes. But this is often not enough. First-order LTL allows the designer to derive properties from a given RBAC policy.

Suppose an RBAC policy is specified as a set of sentences in first-order LTL, and a requirement for this policy is given as another sentence. If we can derive the requirement from the given RBAC policy within first-order LTL, then the requirement in question is fulfilled by this RBAC policy with absolute certainty. In order to reach this certainty, the policy designer has to supply proofs for the requirements as described above. The correctness of the proofs in question can then be verified by Isabelle due to the work described above. Assuming Isabelle works correctly and accepts the proofs, the desired certainty is assured. Since first-order logic is generally undecidable (according to results of Turing [33]), the mentioned proofs can usually not be fully automated, but require human intervention. This applies also to the example of such a proof described in the following subsection.

2) Formal Verification of RBAC Policies: LTL is encoded in Isabelle (cf. `LTL.thy`) as described above. Using LTL as a foundation, we then define RBAC as described in `RBAC1.thy`. The main modification w.r.t. [34] is the introduction of the predicates for delegation named UAO and UAD (see axioms UAO_ax, UAD_ax,$UA2\_ax$). Consequently, the predicate UA is not rigid any more, but UAO now is.

Delegation is defined by an additional Isabelle theory based on RBAC. In this theory, we introduce the predicate $DELEGATE$, and specify delegation axiomatically as defined in Section III-A.2. However, the difference is that we omit the prerequisite conditions for the delegated user for reasons of convenience. In our theory, a user is allowed to delegate a role r if and only if she is assigned to the role r by UAO.

Having specified delegation and revocation exactly, it is now possible to examine the interactions of these concepts with various authorization constraints. This can lead to unexpected results. We now describe an example policy showing an undesirable deadlock behavior under certain conditions. For this purpose, let us assume that delegation and RBAC are defined as described above. Furthermore, we have two distinct users $u1$, $u2$, and an object o with an object-based dynamic SoD condition, i.e., any user is permitted to apply at most one operation to object o. As an additional authorization constraint we demand for reasons of secrecy that any operation [4] can be performed on o at most by one user (i.e., no two distinct users are allowed to apply the same operation to o). Finally, let r be the only role having the permission to apply operation $op1$ to o, and let $u1$ be the only user assigned to r by UAO. Pick now an arbitrary point of time t from which on the following actions happen:

t: User $u1$ delegates role r to $u2$.

$t+2$: User $u2$ applies operation $op1$ to o [5].

$t+3$: User $u1$ revokes role r from $u2$ and and from now on never delegates it again to $u2$.

From this situation we conclude that $UAD(u2, r)$ is true at $t+1$, $t+2$, and $t+3$. However, from $t+4$ and all time steps later $UAD(u2, r)$ is always false. Thus it follows that also $UA(u2, r)$ must be false for those points of time since only for $UAO(u1, r)$ is true, and $u1$ and $u2$ are distinct users. Thus, for those points of time $u2$ will not be allowed to apply operation $op1$ to o since the permission to do this is limited to role r.

By our constraints and prerequisite conditions, we now have the situation that from $t+4$ on no user can apply operation $op1$ to o because user $u2$ cannot do this any more due to revocation and the secrecy constraint forbids access for any other user. Moreover, we can conclude that $u2$ cannot apply any operation to o since she has already performed operation $op1$ on o (object-based dynamic SoD) and once again $op1$ is not available due to revocation. In theorem $Blockade$ this is shown to be true from $t+5$ on. The proof for this has been checked with Isabelle and can be downloaded [32].

Of course, deadlocks are not desirable for an RBAC policy. So let us mention a *positive* example described by Drouineaud et al. [34]. In that scenario, a bank safe is controlled by an IT system, which automatically generates a key, i.e., a secret number. The system then uses a secret sharing scheme to distribute shares of this key to certain users (assigned to the roles cashier and/or director). The distribution

[4]We assume here that an operation reveals some information on the object.

[5]We assume here that till $t+2$ no other user has yet performed $op1$ on o and that $u1$ has not yet revoked role r from $u2$.

process is regulated by SoD (dual control). As can be shown, this RBAC policy meets the requirement to prevent the system from distributing a sufficient number of shares for computing the key to a *single* user. So Isabelle helps to verify security requirements.

3) Advantages of Verification: Proving properties of RBAC policies in first-order LTL with a theorem prover such as Isabelle usually requires human intervention (see above). On the other hand, model checking [35] can be automated. But model checkers only examine a specific model. If a model checker confirms that an RBAC policy and some requirement hold for a given RBAC configuration / model, this need not be true for another RBAC configuration. There may be an RBAC configuration for which the considered RBAC policy holds, but the requirement is not fulfilled. In the worst case, a slight change of a single parameter such as one additional user or object can cause that effect. Therefore one might have to run the model checker for each access demand in order to ensure all given requirements by model checking. At first glance this may seem comfortable, since one would only have to find and define the requirements, and the IT system then could do the rest automatically. Unfortunately, model checking has a high degree of computational complexity, although it is a decidable problem. Hence, the described simple solution for access control may seem advantageous, but would slow down most IT systems to an intolerable extent.

Unlike model checking, deriving properties of RBAC policies in first-order LTL with a theorem prover allows one to give proofs that are independent of the number of users, objects, operations, etc. due to quantification. Assuming the authorization constraints of which the considered RBAC policy consists can easily be implemented, we obtain an IT system that meets all IT security demands for access control in a fast and efficient way. Since a complete automation of verification is generally impossible, we intend to make verification at least more comfortable. Our current research focuses on the following topics:

- Identifying important authorization constraints or combinations of authorization constraints that may serve as building blocks for relevant RBAC policies in areas such as banking via case studies and proving important properties of the discovered constraints or combinations of constraints. The found theorems could then help policy designers who use some of the described elements for building an

RBAC policy and thus save some effort. Indeed, these designers could simply refer to the adequate results without bothering about the proofs.

- Building a library containing the found theorems and proofs, so that the proofs and theorems, respectively, could be reused for further proofs. For example, we could build libraries for delegation, SoD, context constraints, and workflow constraints.
- Investigating the existence of decidable fragments of first-order LTL that may be well-suited for the verification of RBAC policies.

IV. Practical Specification and Validation of RBAC policies with UML and OCL

Having presented the formal specification and verification of RBAC policies, we now turn to a more light-weight specification and validation approach based upon UML and OCL.

A. Specification of RBAC Policies in UML and OCL

First, we briefly explain the basic elements of UML and OCL and thereafter we specify several types of authorization constraints in OCL.

1) UML and OCL: UML [36] is a general-purpose modeling language in which we can specify, visualize, and document the components of software systems. It captures decisions and understanding about systems that must be constructed. UML has become a standard modeling language in the field of software engineering. UML permits to describe static, functional, and dynamic models of software systems. In this paper, we concentrate on the static UML models. A static model provides a structural view of information in a system. Classes are defined in terms of their attributes and relationships. The relationships include specifically associations between classes. In Fig. 2, the static UML model for RBAC consisting of the RBAC classes and associations is depicted (UML class diagram). The classes and associations correspond to the RBAC sets and relations defined in Section II. A further diagram type relevant to our work is the object diagram. Here, objects are instances of classes and links are instances of associations.

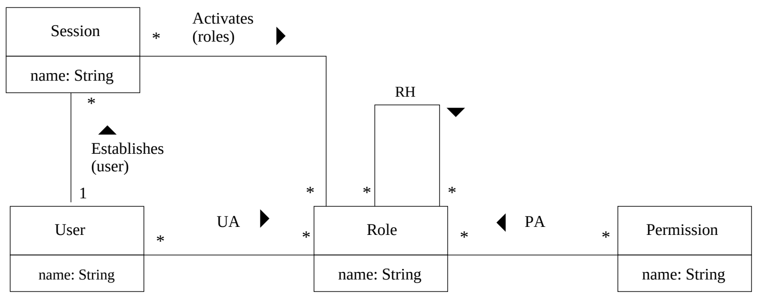


Fig. 2. Class Model for RBAC-Entity Classes.

An object diagram then provides a snapshot of a system at a particular point of time showing objects, their attribute values, and links connecting the objects [36].

OCL [15] is a declarative language that describes constraints on object-oriented models. A constraint is a restriction on one or more values of an object-oriented model. Each OCL expression is written in the context of a specific class. In an OCL expression, the reserved word `self` is used to refer to a contextual instance. The type of the context instance of an OCL expression is written with the `context` keyword, followed by the name of the type. The label `inv:` declares the constraint to be an invariant. Invariants are conditions that must be true during the lifetime of a system for all instances of a given type. The following line shows an example of an OCL invariant describing a role with at most one user:

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

`self` refers to an instance of `Role`. Then `self.user` is a set of `User` objects that is selected by navigating from objects of class `Role` to `User` objects through an association. The "`.`" stands for a navigation. A property of a set is accessed by an arrow "`->`" followed by the name of the property. A property of the set of users is expressed using the `size` operation in this example.

The following shows another example describing that a user can be assigned to a role `r2` only if she is already member of `r1` (prerequisite role constraint introduced by Sandhu et al. [5])[6]:

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

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

The expression `self.role_->includes('r2')` means that `r2` is a member of the set of roles the user is assigned to. The `implies` connector is similar to logical implication. Furthermore, OCL has several built-in operations that can iterate over the members of a collection (set, bag, sequence) such as `forAll`, `exists` `iterate`, and `any` (cf. [15]). These operations are used throughout the rest of the paper.

2) Specification of Authorization Constraints in OCL: In the section before, we have already specified the prerequisite roles constraint in OCL. Subsequently, we give two further examples that demonstrate how to use OCL to specify authorization constraints. The second example shows that even more complex authorization constraints can be formulated in OCL. In fact, we will even prove in Section VI-A that OCL is at least as powerful as the authorization constraint specification language RCL 2000 [6]. As a consequence, our specification and validation approach presented below can deal with all authorization constraints formulated in RCL 2000.

Example 1: Simple Static SoD (SSoD)

The first example concerns an SoD constraint. Consider two conflicting roles such as cashier and cashier supervisor. Mutual exclusion in terms of UA specifies that one individual cannot have both roles. This constraint on UA can be specified using the OCL expression as follows[7]:

```
context User inv SSoD:
let CR:Set={{cashier,cashier_supervisor},{r1, r2},...}
in CR->forAll(cr|cr->intersection(self.role_)->size()<2)
```

This formulation of SSoD is based upon the SSoD specification given by Ahn [6]. Technically, `CR` denotes a set which consists of conflicting role sets.

Example 2: Static SoD - Conflict Users

Even more complex authorization constraints can be formulated in OCL. One example of such a constraint is SSoD-CU identified by Ahn [6]. SSoD-CU (Static SoD - Conflict Users) means that two or more colluding users cannot be assigned to conflicting roles. For example, it might be the company policy that members of the same family cannot be assigned to the roles cashier and cashier supervisor. SSoD-CU

[7]For the sake of simplicity, we have left out here the part for the definition of role instances such as `cashier` and `cashier_supervisor` with the help of OCL's `any` operation, e.g., `cashier:Role=Role.allInstances->any(name='cashier')`. Similar remarks hold for the subsequent OCL specifications.

can now be expressed in OCL in the following way:

```
context User inv SSoD-CU:
let
   CU:Set(Set(User))=Set{Set{Frank,Joe},Set{Sue,Lars}},
   CR:Set(Set(Role))=Set{Set{cashier,cashier_supervisor},...}
in
   CR->forAll(cr|cr->intersection(self.role_)->size()<2)
   and
   CU->forAll(cu|
     CR->forAll(cr|cr->iterate(r:Role; result:Set(User)=Set{}|
     result->union(r.user))->intersection(cu)->size()<2))
```

SSoD-CU is a composite constraint consisting of two parts, an SSoD part and an additional part concerning the conflicting users. The SSoD part is required because otherwise obviously the whole constraint would not be useful. The `iterate` operation iterates over all roles `r` belonging to a set of conflicting roles and collects all users of these roles. `CR` has the same meaning as in Example 1 whereas `CU` is a set consisting of all conflicting user sets.

B. Policy Validation with USE

OCL is a light-weight formalism, which can help in specifying RBAC policies. We now demonstrate how the USE tool [10] is employed for the validation of RBAC policies formulated in UML/OCL. Before describing the validation process in more detail, we first explain the functionality of USE.

1) The USE tool: USE allows the software modeler to validate UML and OCL descriptions and is the only OCL tool allowing interactive monitoring of OCL invariants and the automatic generation of system states. In particular, we use the term “system state” in the following sense:

Definition 4: A *system state* or *snapshot* consists of the current objects and links given by a UML object diagram (cf. Section IV-A.1). A system state must adhere to a UML model, i.e., for each object a class and for each link an association must exist in the corresponding class diagram. A *non-empty system state* at least contains one object.

The central idea of the USE tool is to check for software quality criteria like correct functionality of UML descriptions already in the design level in an implementation-independent manner. This approach takes advantage of descriptive design level specifications by expressing properties concisely and in a more abstract way. Such properties are given by OCL invariants, and these are checked by the USE system against the generated snapshots. These abstract design level tests are expected to be also used later in the implementation phase.

The USE tool expects a textual description of a UML model and its OCL constraints as an input (for an example of such a description refer to Fig. 4). After syntax checks, the model can be displayed by the graphical user interface provided by USE. In particular, USE makes available a project browser which displays all the classes, associations, and invariants of the current model.

Fig. 5 shows a USE screen shot with an example, which is discussed later in Section IV-B.2. On the left, we see the project browser displaying the classes, associations, and invariants. In a detail window below, a selected constraint is pictured. Next to the project browser, we see an object diagram with the current snapshot. The evaluation of the invariants in this system state is pictured in the class invariant window to the right of the object diagram window. The invariant window gives the developer feedback about the validity of the invariants.

The USE tool can now be employed in various ways in the context of RBAC policies (cf. Fig. 3). Specifically, it can be used for the specification (cf. Fig. 4) and for the validation of RBAC policies in the design phase. Validation is the topic of the following section. Furthermore, an authorization engine can be built by using the Java API provided by the USE system. This is discussed in more detail in Section V. The last use case is testing concrete RBAC configurations, i.e., after the deployment of the policy. This aspect is more thoroughly discussed elsewhere [37] and is not topic of this paper.

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

2) Policy Validation: As indicated above, the USE approach to validation is to generate system states and check these states against the specified constraints. In our case, the system states are certain RBAC

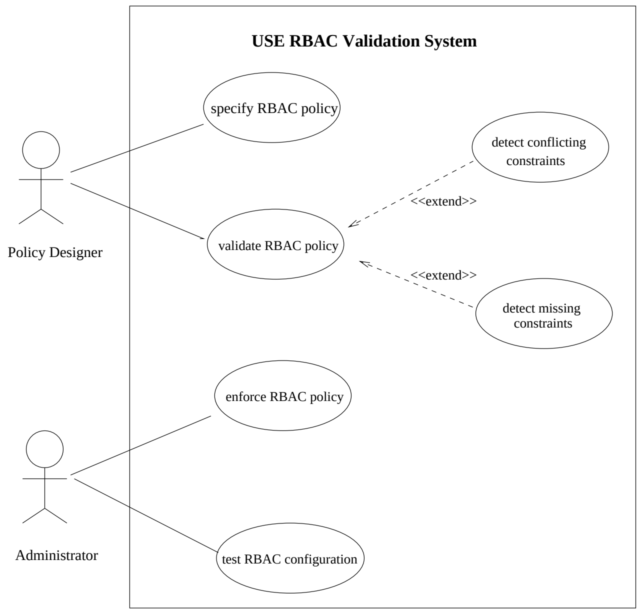


Fig. 3. Use cases for the RBAC USE system.

configurations. The RBAC configurations can be created automatically by running a script with the state manipulation commands, which are supported by the USE tool, or alternatively, with a graphical user interface provided by the USE system [10].

The result of the validation can lead to different consequences. Firstly, we may have reasonable system states that do not satisfy one or more authorization constraints of the policy. This may indicate that the constraints are too strong. Secondly, the RBAC policy may allow undesirable system states, i.e., the constraints are too weak. In the first case, we may have conflicting constraints, whereas in the second case constraints are missing. Subsequently, both situations are discussed more thoroughly. However, we first describe an RBAC policy, which will serve as an example policy throughout the rest of this section.

Example RBAC Policy: The USE specification of the example RBAC policy is depicted in Fig. 4. It consists of the RBAC-related class and association definitions and a set of authorization constraints (cf. Definition 1). The constraints are formulated as OCL invariants. In particular, we define two constraints, one is a prerequisite role constraint between two roles $r1$ and $r2$, the other is an SSoD-CU constraint. However, the static SoD part of the SSoD-CU constraint given in Section IV-A.2 is left out. The reason for this becomes clear when we describe our approach to the detection of missing constraints. Moreover,

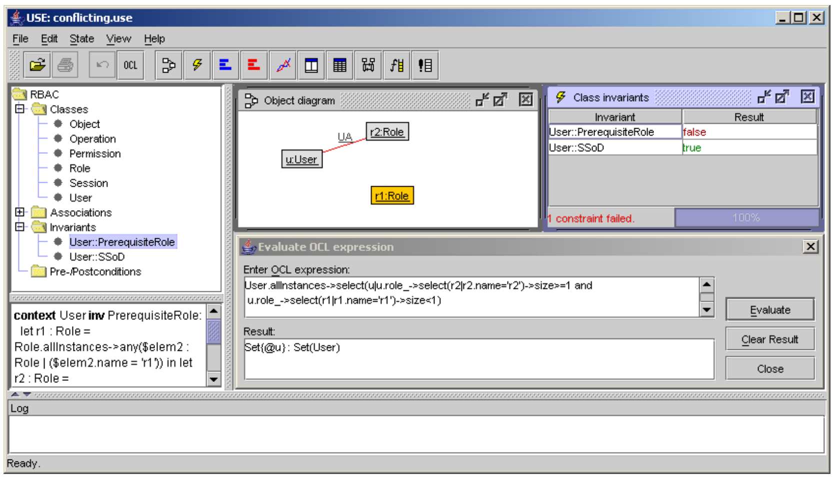
```
model RBAC
--classes

class Role
attributes
name:String
end

class User
attributes
name:String
end

class Permission
attributes
op:Operation
o:Object
end

class Object
attributes
name:String
end

class Operation
attributes
name:String
end

class Session
attributes
name:String
end

-- associations

association UA  between
User[*]  role user
Role[*]  role role_
end
```

```
association PA  between
Permission[*]  role permission
Role[*]  role role_
end

association establishes  between
User[1]  role user
Session[*]  role session
end

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

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

constraints

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

context Role  inv SSoD-CU:
let
CU:Set(Set(User))=Set{{u1,u2,u3},
   {u4,u5}}
in
let
CR:Set(Set(Role))=Set{Set{r1,r2},...}
in
CU->forAll(cu|
 CR->forAll(cr|cr->iterate(r:Role;
 result:Set(User)=oclEmpty(Set(User))|
 result->union(r.user))->
    intersection(cu)->size()<=1))
```

Fig. 4.   USE specification of an RBAC policy.

our example is rather simple for didactic reasons. One should bear in mind that in reality RBAC policies may be considerably more complex, consisting of different kinds of authorization constraints such as various SoD properties, context constraints, and delegation rules. In fact, we also experimented with more complex policies, which contain complex constraints such as object-based static SoD.

Conflicting Constraints: USE may help the policy designer find conflicting constraints as will be demonstrated by means of the aforementioned example. In particular, let us assume that the policy designer has forgotten that she had once defined the prerequisite role constraint between $r1$ and $r2$. Later, the policy designer decided to define $r1$ and $r2$ mutually exclusive due to a change of organizational rules and adds an SSoD constraint between $r1$ and $r2$ to the policy. Obviously, the constraints cannot be satisfied at the same time and hence the composite constraint is too strong. The USE screen shot in Fig. 5 displays the situation after user u has been assigned to $r2$. Clearly, the policy designer cannot assign u to role $r1$; otherwise the new SSoD constraint would be violated. However, now the constraint `User::PrerequisiteRole` is evaluated to false (cf. “Class invariants” view in Fig. 5), and hence the current RBAC configuration is not a correct system state according to the given policy specification, although the configuration seems to be reasonable.



Invariant	Result
User::PrerequisiteRole	false
User::SSoD	true

1 constraint failed. 100%

```
context User inv PrerequisiteRole:
 let r1 : Role =
Role.allInstances->any($elem2 :
Role | ($elem2.name = 'r1')) in let
r2 : Role =
```

Enter OCL expression:

```
User.allInstances->select(u|u.role_->select(r2|r2.name='r2')->size>=1 and
 u.role_->select(r1|r1.name='r1')->size<1)
```

Result:

```
Set{@u} : Set(User)
```

Fig. 5. USE screen shot: two conflicting constraints.

Admittedly, the mere information that a constraint is false might often not help to find the real reason for the problem and to resolve the conflict. Additional information is required which objects and links of the current state violate the constraint. For such a purpose, the policy designer can debug the constraints that are not satisfied by the current system state with the “Evaluate OCL expression” dialog made available by USE. For example, in Fig. 5 the result of the query “all users who are assigned to $r2$ but not to $r1$” applied to the given RBAC configuration is shown. Here, one can learn that u is not assigned to $r1$, although this is required by the prerequisite role constraint. If one now conversely tries to assign u to $r1$, the SSoD constraint fails, and as a consequence one can conclude that the constraints are contradictory.

A policy designer can now employ USE in a similar way for other constraint types such as cardinality constraints or other SoD properties. In particular, this approach is helpful if a new constraint is added to a complex policy, in order to check if it is in conflict with the composition of the already defined constraints, i.e., if at least one of these constraints is evaluated to false.

Nevertheless, USE may find conflicts only in certain cases, and there is no guarantee that all conflicts can be detected. Had u not been assigned to $r2$, the conflict would not have been detected. In order to eliminate contradictory constraints to a larger extent, a more formal approach such as theorem proving is required. On the other hand, the USE approach is only meant to improve the design of an RBAC policy, and does not aim at a formally proven design. Given the condition that there is often a lack of tools for policy analysis, the USE approach can be considered as a first practical step towards more reliable security mechanisms. The formal verification as described in Section III-B and the validation with USE can hence be regarded as complementary.

However, various heuristics can be applied which may streamline the conflict detection process with USE. One approach is sketched in the following. This approach is based upon an automatic snapshot generator which is made available by the USE system [38]. This snapshot generator allows one to define certain properties on the snapshots and to automatically construct sequences of snapshots, apart from manually giving commands or applying the graphical user interface provided by USE. For this purpose, a language called ASSL (A Snapshot Sequence Language) has been introduced such that snapshots can be constructed in a more declarative way. In addition, ASSL has a formal semantics, which has been given by Gogolla et al. [38]. Due to the fact that loop constructs and a backtracking mechanism are provided by ASSL sequences of snapshots can be generated. Moreover, invariants can be dynamically loaded at runtime, either in order to further restrict the snapshots to be constructed (controlling invariants) or to certify given properties. This way, we can rule out trivial snapshots such as the empty snapshot, which obviously satisfies the conflicting constraints mentioned above and does not reveal the conflict.

One remark should be made on the appropriate number of RBAC elements (such as users, roles, user assignment) of which a snapshot should consist. As a rule of thumb, the snapshots should contain the elements occurring in the RBAC policy under investigation. As a consequence, the number of RBAC elements is roughly the sum of the number of elements that occur in the RBAC policy.

The following general recipe can help in detecting conflicting constraints by means of the automatic snapshot generator (assuming that a new constraint is added to a set of non-conflicting constraints):

- specify the set of non-conflicting constraints in a file in USE format,
- define ASSL procedures for generating appropriate entities, attributes and links between the entities (e.g., users, roles, UA and PA relations) and then call these procedures,

- in certain cases, load further controlling invariants dynamically,
- dynamically load the new constraint,
- if there do not exist any snapshots which satisfy all the constraints at the same time, there may be conflicting constraints and a further investigation of the RBAC policy is required.

Note that the aforementioned steps will not necessarily be carried out in the given order. ASSL procedure calls are usually interleaved with dynamically loading further constraints (such as controlling invariants) in order to produce the appropriate snapshot sequences.

In the following, we briefly describe by means of the running example how to use the snapshot generator for the detection of conflicting constraints. In Fig. 6, a protocol file can be found, which contains the central steps for the generation of the snapshot sequences. The aim is to produce snapshots that reveal the aforementioned conflict. First, the RBAC policy in USE format is loaded, containing only a prerequisite role constraint and the RBAC model. Thereafter, appropriate numbers of users and roles are generated, including the roles $r1$ and $r2$. In order to give an impression how ASSL procedures look like, the procedure `generateRoles` is depicted in Fig. 6. This procedure creates roles and checks within a for-loop that no duplicates are generated.

In the next step, the controlling OCL invariant `Role::inv1` is loaded, which guarantees that every user is assigned to at least one role and conversely every role has at least one user. This prevents the creation of trivial snapshots. In the main step, then the new constraint, SSoD, is dynamically added to the policy and six UA links are generated. As a result, we obtain no snapshot which satisfies all the so far defined conditions. This may indicate that there are conflicting constraints, and we can use the query facilities provided by the USE system for a further investigation.

The snapshot generator can also be applied to detect chains of conflicting constraints, consisting of more than two constraints. Here, we can remove one constraint after the other from the USE specification until we obtain a snapshot. The constraint removed in the last step may belong to a conflicting chain. If we remove the constraints in another order, we may identify other constraints of the conflicting chain.

```
use> open Policy.use
use> gen start conflict1.assl generateUsers(3)
use> gen start conflict1.assl generateRoles(4)
use> gen load ControllingInv2.invs
Added invariants:
Role::inv1
use> gen load SSoD.invs
Added invariants:
User::SSoD
use> gen start conflict1.assl generateUA(6)
use> gen result
Checked 4096 snapshots.
Result: No valid state found.
```

```
procedure generateRoles(count:Integer)
var theRoles:Sequence(Role);
begin
theRoles:=CreateN(Role,[count]);
for r:Role in [theRoles]
   begin
     [r].name:=Any([Sequence{'r1','r2','r3','r4'}->reject(n1|Role.allInstances.name->exists(n2|n1=n2))]);
   end;
end;
```

Fig. 6. ASSL commands and an ASSL procedure.

Detection of Missing Constraints: The second consequence of constraint validation may be that a policy permits undesirable system states, i.e., the authorization constraints are too weak. Once again consider the example policy from Fig. 4. If we create a system state, in which u is assigned to the roles $r1$ and $r2$, all constraints (in our case specifically the conflict user part of the SSoD-CU constraint) defined so far are evaluated to true. Hence, the policy seems supposedly to be correct although the policy permits a user being assigned to the mutually exclusive roles $r1$ and $r2$. Obviously, the policy designer has forgotten to define the SSoD part of the SSoD-CU constraint. Therefore, a further SSoD constraint must be added to the policy in order to exclude the undesirable state and to obtain a more restrictive RBAC policy.

In general, we can create snapshots which *deliberately violate* requirements that the RBAC policy under investigation must satisfy. If still all authorization constraints defined so far are fulfilled, then one or more constraints are missing. Moreover, we can also use the aforementioned automatic snapshot generator for detecting missing constraints. This can be done the following way: First, the requirement to be investigated is formulated as an OCL invariant and is at the same time logically negated. For this purpose, a special negate flag can be set with the help of the USE system [38]. By means of the snapshot generator we can then try to create sequences of system states which satisfy both the added (negated) requirement and the already defined constraints. If we find such a system state, one or more constraints are missing.

V. Authorization Engine

In this section, we demonstrate how an RBAC authorization engine can be built based upon the functionality of the USE system. This tool helps to enforce several kinds of authorization constraints like those listed in [6]. A more detailed description of the authorization engine can be found in [39].

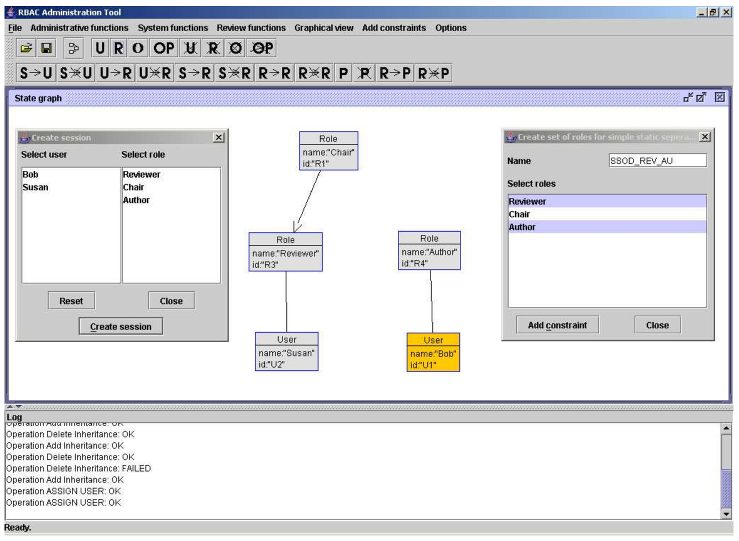
More explicitly speaking, the authorization engine can be used in principle to specify and enforce all authorization constraints expressible in OCL. As a consequence, types of authorization constraints beyond those enumerated by Ahn [6] can also be supported. In the following, the functionality of the authorization engine will be presented. Thereafter, we describe more thoroughly how this tool has been implemented.

A. Functionality of the Authorization Engine

The prototype of the authorization engine currently supports most of the functionality demanded by the RBAC standard [7]. This means that we have implemented administrative functions, system functions, and review functions. *Administrative functions* are required for the creation and maintenance of the RBAC element sets and relations (e.g., UA, PA, RH). For example, *AddUser* and *AssignUser* belong to this class of functions. *System functions* are required by the RBAC authorization engine for session management and making access control decisions. Thus, examples are *CreateSession* and *CheckAccess*. *Review functions* can be employed for inspecting the results of the actions created by administrative functions. Typical examples of review functions are *AssignedUsers* and *UserPermissions*.

Beyond this basic functionality, the RBAC authorization engine provides mechanisms for defining and enforcing role hierarchies and authorization constraints such as various SoD properties, cardinality constraints, prerequisite roles, and context constraints w.r.t. location. In addition, the tool can be quite easily extended to support other authorization constraints. This way, it is flexible enough to enforce various RBAC policies, depending on the internal rules of the organization in question.

To give a better overview, a screen shot of the current prototype of the authorization engine is shown in Fig. 7. In particular, authorization constraints can be defined with the help of dialog windows such as the



Select user	Select role
Bob	Reviewer
Susan	Chair
	Author

Name: SSOD_REV_AU

Select roles: Reviewer, Chair, Author

```
Operation Add Inheritance: OK
Operation Delete Inheritance: OK
Operation Add Inheritance: OK
Operation Delete Inheritance: OK
Operation Delete Inheritance: FAILED
Operation Add Inheritance: OK
Operation ASSIGN USER: OK
Operation ASSIGN USER: OK
```

Fig. 7. The authorization engine.

window on the right-hand side for a static SoD constraint. The current RBAC configuration is visualized by the large window in the middle.

The authorization engine can be employed both at administration- and runtime in order to enforce the aforementioned authorization constraints. For the enforcement of static constraints, an administration tool similar to that depicted in Fig. 7 can be used by a security officer. In case of simple dynamic SoD [18], the session mechanism provided by the considered application can be used: Whenever an application session is generated, a corresponding RBAC session is created in the authorization engine. Similarly, the fact that roles are activated or deactivated can be communicated to the authorization engine. The engine then makes the access decision based upon the application's current security state (cf. Section V-B.4) and hence the authorization engine can be regarded as a policy decision point [40]. One can even integrate the authorization engine with Web services in order to enforce the RBAC policies on the middleware and not on the application level. Then the Web service session can be mapped to an RBAC session.

B. Implementation Aspects

The authorization engine has been implemented by using a Java API made available by the USE tool. This way, the functionality of USE is hidden from the administrator/security officer by the graphical user interface of the authorization engine. Now our implementation, based upon the USE API, will be described in more detail. In particular, we explain how the administrative functions, system functions, and review functions have been realized. Thereafter, the constraint checking mechanism is sketched.

1) Administrative Functions: The core operations provided by the authorization engine are administrative functions. An administrator can change an RBAC configuration with these functions. We have implemented administrative functions by the state manipulation commands of the USE system [10]. To demonstrate this, we subsequently consider the operation $AssignUser$ which assigns a user to a role. $AssignUser$ can be expressed by the state manipulation command `!insert(u,r) into UA` (with a user u and a role r). This command can then be called by employing the command execution facility provided by the USE API called `executeCmd()`. The other administrative functions have been realized in a similar way. Clearly, in order to remain in a state consistent with the current RBAC policy, all (relevant) authorization constraints must also be checked. This will be explained in more detail below.

2) Review Functions: RBAC review functions are demanded by the RBAC standard and have also been conveniently implemented employing the USE functionality. For this purpose, we have employed the query facilities of USE (cf. Section IV-B.2). In particular, the USE API provides the method `eval()`, which evaluates a query consisting of an OCL expression in the current system state. For example, the following OCL query expresses the *UserPermissions* function, which returns all permissions of a user:

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

This query has been specified in a USE file (such as that presented in Fig. 4), which is read when the authorization engine is started. Then the `eval()` method of the expression evaluator provided by the USE system is invoked with the two parameters “UserPermissions” and the user “u”, whose set of permissions is to be determined. The other review functions have been implemented similarly.

3) System Functions: Some RBAC system functions such as *CheckAccess* have been realized with USE similarly to the review functions. As in the case of review functions, the *CheckAccess* function has been specified in the USE file and is then executed by the aforementioned `eval()` method. In contrast, the session-related system functions like *CreateSession* must be realized in the same way as the administrative functions by means of the state manipulation commands.

4) Constraint Checking: The basic idea of the constraint checking mechanism is as follows: The authorization engine checks if the relevant authorization constraints are still satisfied *after* an administrative or system function such as *CreateSession* has been carried out. This is done by the `check()` method made available by the USE API. If any constraint is violated, the last administrative or system function is automatically revoked with the help of an `undo()` method. As a consequence, the tool produces only RBAC configurations that are consistent with the specified RBAC policy.

VI. Discussion and Future Work

There are several directions for further research. We mainly discuss two extensions of our approach that seem to be worthwhile pursuing. First, LTL and OCL are only general-purpose specification formalisms. However, we have not a formalism at hand which is specially *tailored towards* the need of a policy designer. For this reason, we discuss here how RCL 2000 specifications can be translated into OCL statements. The policy designer can then decide to use OCL and/or RCL 2000 for the specification of the RBAC policies. In the latter case, the policy designer can then employ the USE tool for validation and the authorization engine after the translation process.

The second extension of our work concerns history-based constraints: Owing to the fact that USE can only check the current snapshot of a system, history-based authorization constraints cannot be dealt with. For this purpose, a temporal extension of OCL is needed. In the following, we discuss both extensions in more detail. We also compare our approach with other works in a section on related work.

A. Relationship between RCL 2000 and OCL

Given the expressive power of OCL, specification languages for authorization constraints such as RCL 2000 [6] can be translated into OCL. This gives the policy designer the opportunity to specify authorization constraints in a security language and then to validate and enforce the constraints with the help of USE. Subsequently, we show that RCL 2000 expressions have appropriate counterparts in OCL, i.e., we demonstrate that the syntactical constructs of RCL 2000 can be translated into OCL expressions. The reader may be referred to Ahn's thesis [6] in order to obtain more information on the details of RCL 2000's syntax.

RCL 2000 sets and relations: The basic RBAC sets such as U, R and P are modeled by UML classes. The relations such as UA, PA, and RH can be represented by the UML associations presented in Fig. 4. In addition, the sets CU, CP, and CR are expressed by local variables in a `let` statement (cf. Section IV-A.2). Although OCL only supports finite sets, this is no problem here because RCL 2000 itself only supports finite sets.

Operators: Operators like $\Rightarrow$, $\wedge$, $\leq$ or $\geq$ have their obvious counterparts in OCL, namely, `implies`, `and`, $\leq$ or $\geq$. $|X|$ returns the cardinality of a set X and can be expressed by OCL's `size` operator.

RBAC functions: RCL 2000 supports various RBAC functions. We now demonstrate by means of several examples that the RBAC functions can be modeled in OCL. Due to space limitations we do not cover every RBAC function here. One example is the $user$ function, which returns the unique user of a session. This function can be trivially represented by `self.user` if the context of the OCL expression is `Session`. The overloaded $roles$ function returns all the roles belonging to a user, session, and permission, respectively. Moreover, $roles^*$ is a variant of $roles$ which additionally takes the role hierarchy into consideration. For example, $roles^*(u) = \{r \in \ R \mid \exists\, r1 \in R.\ r \leq r1 \wedge (u, r1) \in UA\}$ returns all the roles assigned to user u and all the roles junior to them. $roles^*$ can be formulated in OCL as follows:

```
roles_star(u:User):Set(Role)=
   Role.allInstances->select(r|r.senior->exists(r1|u.role->includes(r1))).
```

Note that `r.senior` by definition already contains all the roles senior to `r` due to the transitivity of $\leq$. Hence, we need not specify a function which calculates the senior roles of `r` recursively.

Non-deterministic functions: The central functions of RCL 2000, however, are OE (one element) and AO (all other), which are both non-deterministic functions. $OE(X)$ selects non-deterministically one element from a set X. Multiple occurrences of the expression $OE(X)$ within an RCL 2000 statement return always the same element of X. $AO(X)$ gives then the complement $X - \{OE(X)\}$ on any set X. Hence, we have the equation $X = \{OE(X)\} \cup AO(X)$.

The OE function can be expressed by the `any` operation provided by OCL with the condition set to `true`, i.e., $OE(X)$ corresponds to `X->any(true)`. As mentioned above, multiple occurrences of the $OE(X)$ expression always yield the same element. This can be conveniently expressed by factoring out the `any` expression in a `let` expression. $AO(X)$ can then be translated to `X-{X->any(true)}`.

For example, consider the following RCL 2000 expression containing OE and AO terms:

$$OE(OE(CR)) \in roles(OE(U)) \Rightarrow AO(OE(CR)) \cap roles(OE(U)) = \emptyset$$

Then the corresponding OCL expression is:

```
let
  CR:Set(Set(Role))={...},
  cr:Set(Role)=CR->any(true),
  r:Role=cr->any(true),
  u:User=User.allInstances->any(true)
in
  u.role_->includes(r) implies((cr-{r})->intersection(u.role_)={})
```

Having shown that the basic elements of RCL 2000 can be converted to OCL, we now briefly *sketch* a translation algorithm: In the first step of this algorithm all the AO expressions are eliminated from the RCL 2000 expression. Then we iteratively translate OE terms into OCL expressions, introducing new local variables in a `let` construct as explained above. This way, the same occurrences of OE expressions within an RCL 2000 expression can be factored out. If we have nested OE expressions, the translation will be started from the innermost OE term.

The analysis of the runtime depends on the number of OE terms. Therefore, this algorithm can translate an RCL 2000 expression in $O(n)$, supposing that n is the number of the different OE terms. As future work a compiler could be developed that parses the RCL 2000 statements and converts them into OCL expressions, based upon the aforementioned algorithm.

B. History-based constraints

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

In the following, we sketch how history-based authorization constraints can be specified in TOCL (Temporal OCL) [41], an extension of OCL with temporal elements. In particular, temporal operators like `always` (in the future), `sometime` (in the future), and `next` are available. To demonstrate how history-based authorization constraints can be formulated in TOCL, we take object-based dynamic SoD as an example. In order to specify object-based dynamic SoD, we use the the predicate $Exec(u, op, obj)$ introduced in Section III-A.1. However, due to the fact that $Exec$ is a ternary predicate and OCL does not directly support ternary associations[8] we extend (T)OCL with an additional predicate `Exec` to express ternary associations, as proposed by Gogolla et al. [42]. Then, we obtain the following TOCL specification for object-based dynamic SoD:

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

This corresponds to the first-order LTL specification given in Section III-A.1:

$$\forall u : User;\ op, op1 : Operation;\ obj : Object.op \neq op1 \wedge Exec(u, op, obj) \Rightarrow \Box\neg Exec(u, op1, obj)$$

Having a formalism for the specification of history-based constraints at hand, the next step would be to extend USE itself in order to support TOCL. This way, the authorization engine could also enforce history-based constraints, which are often required in the context of workflows [3].

In this respect, another direction for future work would be to develop a compiler which translates TOCL specifications into first-order LTL. Then we could utilize Isabelle to formally verify properties of the LTL representation (and indirectly of the TOCL specification) of the RBAC policy in question. To sum up, we

[8]Strictly speaking, with the help of association classes ternary associations can be expressed in OCL, but this construction is a bit clumsy.

	RCL 2000	UML/OCL	LTL	TOCL
Static SoD constraints	yes	yes	yes	yes
Simple Dynamic SoD [18]	yes	yes	yes	yes
History-based SoD constraints	no	no	yes	yes
Workflow constraints	no	no	yes	yes
Cardinality constraints	yes	yes	yes	yes
Prereq. roles/permissions	yes	yes	yes	yes
Context constraints w.r.t. location	no	yes	yes	yes
Context constraints w.r.t. time [21]	no	currently no support	currently no support	currently no support
Delegation/revocation	no	yes	yes	yes
Verification/validation	no support	validation with USE	formal verification with Isabelle	currently no support
Authorization engine	currently no support	yes	currently no support	currently no support

TABLE I

OVERVIEW OF RBAC SPECIFICATION LANGUAGE PROPERTIES.

could now enforce dynamic RBAC policies with the enhanced USE system and at the same time verify dynamic RBAC policies by means of Isabelle. This way the gap between the (T)OCL and first-order LTL approach can be filled. As a summary, Tab. I presents the essential properties of the RBAC specification languages discussed in this paper.

C. Related Work

There are several works on the formal specification of authorization constraints such as [21], [6], [8], [13]. Gligor et al. [13] formalize history via traces of states, but end up with rather complex formulas explicitly talking about states. In contrast, first-order LTL allows for an elegant formulation of history-based SoD constraints.

Joshi et al. define the GTRBAC model which has the notions of temporal constraints and events [21]. Joshi et al. explicitly introduce points of time and duration in order to specify temporal contexts whereas we currently concentrate on history-based and order-based SoD constraints. Clearly, we can extend our library of Isabelle theories in order to support GTRBAC and could introduce predicates for events. This is needed, for example, if we would like to deal with temporal delegation constraints such as duration constraints on delegation rules. In addition, Shafiq et al. present a verification framework using Petri nets, which is based on GTRBAC [43]. Similarly to the validation approach with USE, the framework can detect conflicting and missing constraints. In contrast to the USE approach, the verification is tailored to GTRBAC's event-based model. However, this verification approach can handle only finite sets of RBAC entities and has exponential time and space complexity. In contrast, we can prove general theorems on RBAC policies by means of theorem proving with Isabelle, regardless whether the underlying sets are infinite or change at runtime.

Often the graph-based approach presented by Koch et al. [44] is mentioned in the context of policy verification. Among other aspects, this approach ensures that concrete RBAC configurations remain consistent w.r.t. a specified RBAC policy when administrative RBAC functions (represented as graph rules) are performed. In this respect, this approach is similar to our enforcement mechanism based upon USE. In fact, one could also build an authorization engine based upon a graph transformation engine. In addition, Koch et al. present an approach to the detection and resolution of conflicting constraint pairs, similar to the validation with USE. Currently, however, no tool support for the detection of conflicting constraints seems to exist.

Crampton presents another authorization engine [45]. However, with Crampton's approach, for example, neither the SSoD-CU constraint nor context constraints are supported. On the other hand, no history-based SoD constraints can be currently enforced with the USE approach.

In this paper, we only presented a simple delegation and revocation scheme. In fact, newer schemes [28], [46], [29] could also be expressed by means of the LTL approach. For example, Atluri et al. extended the notion of delegation to allow conditional delegation, where the delegation conditions can be based on time, workload and task attributes, specifically focusing on constraints associated with workflow systems [28]. In addition, GTRBAC has recently adopted role-based delegation notions with hybrid hierarchies and multiple hierarchy semantics to support fine-grained delegation schemes [29].

VII. Conclusion

In this paper, we presented a methodology for the specification, verification, and enforcement of RBAC policies. In particular, we demonstrated that several types of authorization constraints can be specified with the help of the formalisms first-order LTL, OCL and TOCL. Due to the fact that UML/OCL is quite familiar in industrial environments there is hope that OCL can be used by policy designers in many organizations. In addition, we demonstrated that the theorem prover Isabelle can help in formally verifying properties of complex dynamic RBAC policies. The validation approach with the USE system, however, can be regarded as a first line of defense, which can be used to fulfill several practical needs. First, USE can be employed to validate authorization constraints. This way, certain conflicts between authorization constraints and missing constraints can be detected. Second, the Java API provided by USE can be utilized to build an authorization engine which helps to enforce various RBAC policies.

To sum up, both the formal and the practical approach to policy specification and verification can be used in the design phase in order to rule out inconsistencies and undesirable properties of RBAC policies early. Last but not least, the approaches can be regarded as complementary rather than competitive.

References

[1] KPMG, “Fraud survey reports 1996-2002,” 2006, KPMG International Canada.

[2] EU, “Directive on the protection of individuals with regard to the processing of personal data and on the free movement of such data. Directive 95/46/EC. http://www.privacy.org/pi/intl_orgs/ec/eudp.html,” 1995.

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

[4] D. D. Clark and D. R. Wilson, “A comparison of commercial and military computer security policies,” *Proc. of the 1987 IEEE Symposium on Security and Privacy*, pp. 184–194, 1987.

[5] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, “Role-based access control models,” *IEEE Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[6] G.-J. Ahn, “The RCL 2000 language for specifying role-based authorization constraints,” Ph.D. dissertation, George Mason University, Fairfax, Virginia, 1999.

[7] American National Standards Institute Inc., “Role Based Access Control,” 2004, ANSI-INCITS 359-2004.

[8] T. Jaeger and J. Tidswell, “Practical safety in flexible access control models,” *ACM TISSEC*, vol. 4, no. 2, pp. 158–190, May 2001.

[9] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[10] M. Richters, “A Precise Approach to Validating UML Models and OCL Constraints,” Ph.D. dissertation, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[11] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[12] R. Goldblatt, *Logics of Time and Computation, Second Edition, Revised and Expanded*, ser. CSLI Lecture Notes. CSLI, Stanford, 1992 (first edition 1987), vol. 7, distributed by University of Chicago Press.

[13] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo, “On the formal definition of separation-of-duty policies and their composition,” in *1998 IEEE Symposium on Security and Privacy (SSP ’98)*. IEEE, May 1998, pp. 172–185.

[14] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer-Verlag, 1992.

[15] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

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

[17] I. Ray, N. Li, R. France, and D.-K. Kim, “Using UML to visualize role-based access control constraints,” in *Proc. of the 9th ACM symposium on Access control models and technologies*. ACM Press New York, USA, 2004, pp. 115–124.

[18] R. Simon and M. Zurko, “Separation of duty in role-based environments,” in *10th IEEE Computer Security Foundations Workshop (CSFW ’97)*, June 1997, pp. 183–194.
[19] K. Sohr, M. Drouineaud, and G.-J. Ahn, “Formal Specification of Role-based Security Policies for Clinical Information Systems, Santa Fe, New Mexico,” in *Proc. of the 20th ACM Symposium on Applied Computing*, 2005.
[20] V. Atluri and J. Warner, “Supporting conditional delegation in secure workflow management systems.” in *SACMAT*, E. Ferrari and G.-J. Ahn, Eds., 2005, pp. 49–58.
[21] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor, “A generalized temporal role-based access control model.” *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 1, pp. 4–23, 2005.
[22] E. Bertino, E. Ferrari, and V. Atluri, “The specification and enforcement of authorization constraints in workflow management systems.” *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 1, pp. 65–104, 1999.
[23] T. Mossakowski, M. Drouineaud, and K. Sohr, “A temporal-logic extension of role-based access control covering dynamic separation of duties,” in *Proc. of TIME-ICTL 2003, Cairns, Queensland, Australia*, July 8–10 2003.
[24] R. Sandhu, “Transaction control expressions for separation of duties,” *Fourth Aerospace Computer Security Applications Conference, Orlando*, pp. 282–286, 1988.
[25] A. Schaad, V. Lotz, and K. Sohr, “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, June 2006.
[26] E. Barka and R. Sandhu, “A role-based delegation model and some extensions,” in *Proc. of 16th Annual Computer Security Application Conference*, Dec. 11–15 2000, pp. 125–134.
[27] H. M. Gladney, “Access control for large collections,” *ACM Trans. on Information Systems*, vol. 15, no. 2, pp. 154–194, 1997.
[28] V. Atluri and J. Warner, “Supporting conditional delegation in secure workflow management systems,” in *Proc. of the 10th ACM Symposium on Access Control Models and Technologies*, Stockholm, Sweden, June 1–3 2005, pp. 49–58.
[29] J. Joshi and E. Bertino, “Fine-grained role-based delegation in presence of the hybrid role hierarchy,” in *Proc. of the 11th ACM Symposium on Access Control Models and Technologies*, Lake Tahoe, California, USA, June 7–9 2006, pp. 81–90.
[30] L. Zhang, G.-J. Ahn, and B.-T. Chu, “A rule-based framework for role-based delegation and revocation,” *ACM Transactions on Information and System Security*, vol. 6, no. 3, pp. 404–441, Aug. 2003.
[31] A. Hagström, S. Jajodia, F. Parisi-Presicce, and D. Wijesekera, “Revocations – a classification,” in *14th IEEE Computer Security Foundations Workshop (CSFW ’01)*, June 2001, pp. 44–58.
[32] K. Sohr and M. Drouineaud, “Isabelle Theories. http://www.sis.uncc.edu/liisp/rbac/Isabelle.zip,” 2005.
[33] A. Turing, “On Computable Numbers with an Application to the Entscheidungsproblem,” *Proceedings of the London Mathematical Society (2)*, vol. 42, pp. 230–265, 1936. [Online]. Available: `www.abelard.org/turpap2`
[34] M. Drouineaud, M. Bortin, P. Torrini, and K. Sohr, “A first step towards formal verification of security policy properties for RBAC,” in *Proc. of the 4th International Conference on Quality Software*, 2004, pp. 60–67.
[35] E. Clarke, O. Grumberg, and A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.
[36] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual, Second Edition*, ser. Object Technology Series. Reading, Mass.: Addison Wesley Longman, 2004.
[37] K. Sohr, G.-J. Ahn, M. Gogolla, and L. Migge, “Specification and validation of authorisation constraints with UML and OCL,” in *Proc. of the 10th European Symposium on Research in Computer Security*, 2005.
[38] M. Gogolla, J. Bohling, and M. Richters, “Validation of UML and OCL Models by Automatic Snapshot Generation,” in *Proc. 6th Int. Conf. Unified Modeling Language (UML’2003)*. Springer, Berlin, LNCS 2863, 2003, pp. 265–279.
[39] K. Sohr, G.-J. Ahn, and L. Migge, “Articulating and enforcing authorisation policies with UML and OCL,” in *Proc. of the ACM ICSE Workshop on Software Engineering for Secure Systems (SESS05)*, St. Louis, MO, 15-16 May 2005.
[40] OASIS, “eXtensible Access Control Markup Language (XACML) Version 2.0,” 2005. [Online]. Available: `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`
[41] P. Ziemann and M. Gogolla, “An OCL Extension for Formulating Temporal Constraints,” Universität Bremen, Res. Report 1/03, 2003.
[42] M. Gogolla and M. Richters, “Transformation Rules for UML Class Diagrams,” in *Proc. 1st Int. Workshop Unified Modeling Language (UML’98)*. Springer, Berlin, LNCS 1618, 1999, pp. 92–106.
[43] B. Shafiq, A. Masood, J. Joshi, and A. Ghafoor, “A role-based access control policy verification framework for real-time systems,” in *Proc. of IEEE Workshop on Object-oriented Real-time Databases*, 2005, pp. 13–20.
[44] M. Koch, L. Mancini, and F. Parisi-Presicce, “Graph-based specification of access control policies,” *Journal of Computer and System Sciences*, vol. 71, no. 3, pp. 1–33, 2005.
[45] J. Crampton, “Specifying and enforcing constraints in role-based access control,” in *Proc. of the 8th ACM Symposium on Access Control Models and Technologies*. New York: ACM Press, June 2–3 2003, pp. 43–50.
[46] J. Wainer and A. Kumar, “A fine-grained, controllable, user-to-user delegation method in RBAC,” in *Proc. of the 10th ACM Symposium on Access Control Models and Technologies*, Stockholm, Sweden, June 1–3 2005, pp. 59–66.