ORKA – Organisational Control Architecture

ORKA

Deliverable AP 4.1

Formalisms and Methods for the Validation of RBAC Policies

On Classifying Validation Methods

31.12.2006

Michael Drouineaud, TZI / Universitaet Bremen
Karsten Sohr, TZI / Universitaet Bremen
Christopher Alm, HITeC / University of Hamburg

ORKA is funded by the German Ministry of Education and Research (BMBF) as part of its Software Engineering 2006 programme.

Federal Ministry
of Education
and Research

© 2006 ORKA Consortium

Internal document information:

`$Id: del-ap4.1.tex,v 1.16 2007/02/20 15:21:05 alm Exp $`

Contents

1 Validation for RBAC security policies **4**
1.1 Introduction 4
1.2 Foundations 5
1.2.1 Example 6

2 Formalisms **6**
2.1 First-order LTL 6
2.1.1 Propositional LTL 7
2.2 Graph-based Formalism for RBAC 8
2.3 DTAC model, Entity-Relationship model 10
2.4 UML, OCL, TOCL 11

3 Supporting tools for Validation **12**
3.1 The Theorem Prover Isabelle 12
3.2 The Model Checker NuSMV 13
3.2.1 Model Checking RBAC Configurations for Workflows 14
3.2.2 Analysis of the Interplay between SoD Properties and Delegation . . . 15
3.2.3 Inter-Instance Analysis of Workflows 15
3.2.4 Authorization Engine for RBAC Policies 16
3.3 Alcoa 16
3.4 Validation with Graph Transformation Tools 18
3.5 Specification and Validation with UML and OCL 18
3.5.1 The USE System 20
3.5.2 Validation of RBAC Policies 21
3.5.3 Authorization Engine 21
3.5.4 Testing a Given RBAC Configuration with USE 22

4 Assessment of the considered formalisms and tools **23**
4.1 Formalisms 23
4.1.1 (First-order) LTL 24
4.1.2 Graphs 25
4.1.3 DTAC model, Entity-Relationship model 25
4.1.4 UML,OCL,TOCL 26
4.2 Tools 26
4.2.1 Isabelle 26
4.2.2 NuSMV 27
4.2.3 Alcoa 27
4.2.4 AGG 28
4.2.5 The USE tool 28
4.3 Summary 28

1 Validation for RBAC security policies

1.1 Introduction

Since the relevance of IT systems for authorities and enterprises is expanding constantly, the importance of IT security is increasing rapidly. This means IT security measures for protecting data inside such systems and/or enforcing authorization constraints such as separation of duty for business processes are becoming indispensable. This holds all the more, on the account that formerly separated IT systems are integrated (c.f. Web services) and that in general the complexity of IT systems is steadily increasing. Furthermore, high levels of IT security standards such as Common Criteria demand the application of formal validation methods to 'select TOE security policies' (TOE = target of evaluation).
Finally, it is a well-known fact that conceptual faults are very hard to correct after the implementation of the IT system, as may be seen from the following examples:

- When extending credit cards with RFID capabilities, the designers apparently did not pay much attention to security. Consequently, data like credit card number, cardholder name, etc. were at least in some cases stored as plaintext in the RFID chip attached to the credit card. Thus a common RFID scanner suffices to read these data from an appropriate distance. Most of the cardholders will probably not feel very comfortable with this risk.
- As WLAN standards had to be decided, the intended cryptographic methods were already obsolete and additionally implemented such a way that they could be broken very easily, which made security experts disadvise companies to use WLAN.
- A large travel agency offered its clients the possibility of online-booking. For this purpose, the clients had to enter their address and so on in a web form. During this process the client would also pick a password which would be required for later modifications in order to identify this specific client. Unfortunately the data to be handed over at the first contact did not contain any secret (e.g. a public key) ensuring the identity of the client. In fact for many persons this data can be easily retrieved from the web. Thus a client might easily deny his participation in a transaction.

Validation in general helps to avoid such mistakes. The above examples also demonstrate that validation should be applied at the beginning of the design process (as IT security principles in general) because thorough understanding of the uses and abuses of a system is the first step toward economical and effective security. The example in subsection 1.2.1 indicates, how validation can help to make assertions about RBAC security policies.
This deliverable presents and evaluates formalisms and tools for the validation of RBAC security policies. Among other things it contains a very detailed description of the employment of the model checker NuSMV for the validation of security requirements for workflows due to recent publications in this area, which is demonstrated by an example. Furthermore, the application of the USE Tool for the validation is discussed extensively. Moreover, as demonstrated in a recent master thesis, USE Tool can be the core of an authorization engine.
The deliverable consists of four sections. The first section will introduce basic notions and give a short example for validation of RBAC security policies, the second will describe various (more or less) helpful formalisms for validation, the third section will describe some tools and the fourth section will give an assessment of the presented formalisms and tools w.r.t. validation.

1.2 Foundations

The following subsection aims at providing the essential descriptions for the understanding of validation. Since there is no predetermined formalism, mathematical definitions are not given here. The first two descriptions characterize the objects of validation. It is presumed that the reader is familiar with the concepts and notions of role-based access control (RBAC) (cf. [SCFY96, Ame04]).

Description 1 *A **regular** RBAC configuration* [1] *consists of finite sets of users, objects and roles with explicitly specified operations and state transitions such that the access behavior of the system is unequivocally determined. A (regular) RBAC configuration may be a (role-based) workflow instance, a Kripke model (for linear temporal logic) in case of model checking, a sequence of graphs (where the successor can be obtained from the predecessor via graph transformation) or a similar structure. Furthermore, a regular RBAC configuration is required to be finite in the sense, that after the execution of a finite number of access operations a terminal state is reached (i.e. no further access operation is executed) or all further access operations can be described by a periodic pattern. Thus there is a terminating deterministic algorithm which can decide whether e.g. an arbitrary sequence of access operations on certain objects takes place or not.*

Consequently RBAC configurations:

- having an infinite set of users, objects or roles or
- being not finite in terms of access behavior as specified above

are denoted **irregular** RBAC configurations.

Description 2 *An RBAC (security) policy is a finite set of authorization constraints and other schematic rules. Thus an RBAC (security) policy in general corresponds to a schema defining an infinite set of RBAC configurations. In formalisms like first-order linear temporal logic such a policy usually consists of a finite set of closed formulas* [2].

The following descriptions characterize the two kinds of validation, which are considered in ORKA.

Description 3 *Configuration-based respectively automated validation denotes checking requirements of an RBAC security policy such as authorization constraints for a **finite** set of regular RBAC configurations by a terminating deterministic automated algorithm. By the essential properties of **regular** RBAC configurations such an algorithm can always be found. Configuration-based or automated validation mostly employs tools like model checkers or constraint solvers, which are rather independent of human intervention.*

The reader may notice that finding a (non-trivial) RBAC configuration for which a RBAC security policy holds ensures the consistency of this security policy. To be a little more precise, if we define an RBAC policy as a finite set of (closed) formulas of first-order or higher-order (linear temporal) logic and can produce some (regular) non-trivial RBAC configuration for which

[1] The reader may notice that in workpackage 2.2 the notion 'RBAC configuration' is defined diversely (without state transitions).

[2] For reasons of convenience we demand that an RBAC security policy can always be expressed as a finite set of closed formulas in formalisms like first-order or higher-order (linear temporal) logic.

this set of formulas holds, then we know by fundamental theorems of logic that the considered set of formulas is contradiction-free. This result can be easily obtained because the RBAC configuration is a model for the set of formulas in question.

Description 4 *Proof-based respectively deduction-based validation means deriving security relevant properties of RBAC (security) policies via adequate inference rules. Such properties hold in general for an infinite set of RBAC configurations (cp. description 2). Hence this kind of validation usually requires human intervention. Common formalisms for proof-based or deduction-based validation use first-order or higher-order logic. In ORKA specifically linear first-order temporal logic is intended as a formalism for this kind of validation. This formalism has been encoded in the theorem prover Isabelle/HOL (cf. [DBTS04]).*

The reader may notice that the properties derived from an RBAC policy by proof-based validation hold for all RBAC configurations defined by the corresponding schema including **irregular** RBAC configurations.
Altogether, configuration-based and proof-based validation can be regarded as complementary rather than competing methods for the validation of RBAC security policies. As mentioned above, configuration-based validation allows to prove the consistency of an RBAC security policy and proof-based validation can derive (security relevant) properties of an RBAC security policy, which hold for all RBAC configurations defined by the corresponding schema.
The following text shortly presents an example for the application of proof-based validation, which has been published at QSIC 2004 [DBTS04].

1.2.1 Example

Suppose a bank safe is controlled by an IT system which requires a secret number to open the safe. For this reason, users authorized for certain RBAC roles can get shares of that secret number generated via a secret sharing scheme. It takes three different shares to compute the secret number. The distribution of shares is governed by the following RBAC policy:
Any user entitled to assume the role bank director shall be able to execute the operation for getting shares twice but no more, any user entitled to assume the role cashier is allowed to execute the operation once but no more (we assume, that no two users get the same share and no user gets the same share twice). Furthermore, no user shall be able to get a share without being entitled to assume the role director or cashier.
From this we can deduce that the policy does not allow any single user to obtain three different shares (without the help of another user), which has been verified by the theorem prover Isabelle [DBTS04].
The following section describes formalisms which can help proving and/or examining properties of RBAC security policies.

2 Formalisms

2.1 First-order LTL

Proof-based validation can be achieved by using first-order linear temporal logic (first-order LTL) as formalism to prove certain properties of RBAC security policies. Such properties (ex-

pressed as first-order LTL formulas) typically hold for a class of models respectively RBAC configurations not merely for a single (regular) RBAC configuration. First-order LTL has been embedded in the theorem prover Isabelle/HOL [PN] as theory (cf. description 4 in subsection 1.2). Isabelle helps to assure the correctness of the proofs for the lemmata respectively theorems. Obviously first-order LTL is a solid very expressive [3] formalism with mathematically defined syntax and semantics (Kripke semantics [Gol87]). 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 $\ominus$. Models live over discrete time, indexed by the natural numbers as time steps. First-order LTL can among other things express any authorization constraint which can be specified in the formal language RCL 2000 [Ahn99] since RCL 2000 is equivalent to a restricted form of first-order logic (cp. [AS00]). Furthermore, dynamic authorization constraints like Object-based Separation of Duties (ObjDSoD) can easily be specified because first-order LTL is a temporal logic. Followingly, ObjDSoD is shortly described and then specified in first-order LTL in order to provide an example of the use of first-order LTL as formalism.
According to ObjDSoD a user may be entitled to execute several operations, but she may not be allowed to apply two or more of those operations to the same object. This can be expressed in first-order LTL in the following way:

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

The predicates ‘Exec’ and ‘Auth’ indicate the application of an operation to an object by a user respectively the authorization of a user to do such a thing. The above formula corresponds to the statement, that any user who applies the operation *op* to a given object *obj* can in that moment and from that moment on never be authorized to apply any other operation to the same object. This constraint may, for example, apply to operations like **create_order** and **confirm_order** (w.r.t. the same order) for many organizations like companies, authorities and universities.
It is intended to reuse lemmata that have been proven for certain combinations of authorization constraints in order to prove theorems about more complex RBAC security policies which use such combinations as building blocks. Thus first-order LTL can help to assure properties of RBAC security policies that are too complex to be easily understood by human beings.
Proving security properties in first-order LTL may be a good method to meet the requirements for certifications of high-level security according to IT security standards like Common Criteria. If a single regular RBAC configuration or a finite set of regular RBAC configurations is to be examined, then propositional LTL is an appropriate formalism as described below.

2.1.1 Propositional LTL

Let an RBAC workflow with its resource model (e.g., user-role assignment, user-task instance assignment, permission assignment) be given. Does this RBAC configuration/resource model of the workflow adhere to specified security properties such as object-based dynamic SoD, history-based SoD, prerequisite role constraints, or cardinality constraints? A typical example of a violation of such a security property would be a clerk may prepare and approve a

[3] Indeed, first-order logic is a sufficient foundation for many parts of mathematics (including set theory).

check. Moreover, if delegations and revocations of task instances or authorizations are carried out during the execution of a workflow (instance), are the aforementioned security properties satisfied? In particular, we can regard the workflows as reactive systems [MP95] where certain tasks and delegations are performed at runtime. Since workflows (for example, due to loops and branches) can be quite complex, an automated analysis of such (regular) RBAC configurations for workflows is desirable. Specifically, the automated analysis should be able to check the aforementioned temporal/dynamic properties. In addition, can the analysis tool also be employed to implement an authorization engine that helps in enforcing the dynamic access control policies? In order to aid in the automated analysis of complex reactive systems and properties as described before, we can apply model-checking techniques [CGP99]. Such techniques have already been used and refined in other domains such as safety-critical systems analysis, e.g., to verify the correctness of railway control systems or aircraft controllers. Model checking is a technique for the automated verification of *finite* state-based (concurrent) systems. The proof of a property is entirely carried out by the machine. In case the property does not hold, the model checker will construct a counter-example suitable for failure diagnosis. In mathematical terms, the considered (finite) systems are represented as finite state-based transition graphs (Finite State Machine, FSM). A *Finite State Machine* consists of a finite set of states; a set of initial states (a subset of the set of states); a transition relation (states are accessible from the current state); a function mapping each state to the atomic propositions holding in this state. These FSMs are often referred to as Kripke models [CGP99]. The aim of model checking is to automatically verify that the FSM in question satisfies certain properties. The reader may notice that a non-trivial regular RBAC configuration for an RBAC policy (cp. subsection 1.2) is a FSM. Thus model checking may be employed for checking the consistency of a given RBAC policy.

A FSM can be specified with the help of the SMV input language. However, we also need a way to specify the properties which the FSM should satisfy. NuSMV offers two formalisms for this purpose, namely propositional CTL (computation tree logic) and propositional LTL. In Section 3.2.1, we will use LTL for the specification of dynamic SoD properties. As pointed out [MDS03], LTL is well-suited to specifying dynamic access control policies.

LTL [Gol87] uses the familiar Boolean operators like $\wedge$ and $\vee$. On the other hand, special temporal operators have been introduced:

- `F p` (sometimes in the future holds `p`),
- `G p` (globally in the future holds `p`),
- `p U q` (`p` holds until `q`), and
- `X p` (`p` is true in the next step).

Moreover, corresponding past modalities are also available (such as `H` - historically, `O` - once in the past, `Y` - one step before). To sum up, LTL characterizes each linear path induced by an FSM. NuSMV allows one to specify properties in an extra section called LTLSPEC. It is possible to define several LTL properties for an FSM at the same time.

2.2 Graph-based Formalism for RBAC

In this subsection, we present the graph-based formalism introduced by Koch et al. in more detail as another specification format for RBAC policies. A graph-based RBAC configuration

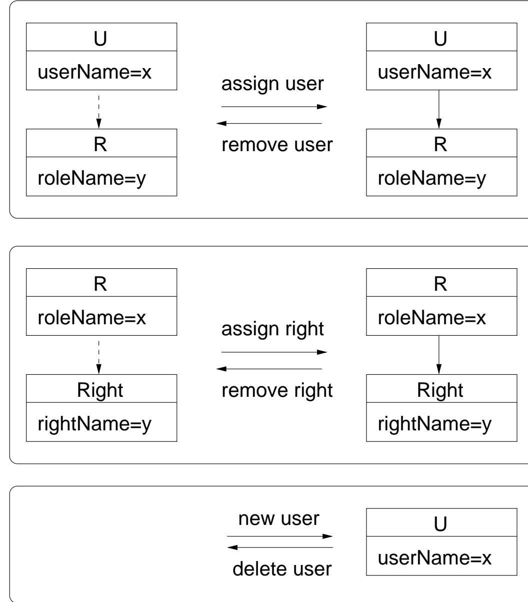

Figure 1: Examples of RBAC rules.

state consists of an initial graph representing the initial protection state of the system (= RBAC configuration state) and a set of *graph rules* to describe the dynamic changes of the protection state.

We introduce next the notions of a graph and a graph rule. For a general introduction to graph transformations see [KMPP02]. A *graph* G consists of disjoint sets of *nodes* G_N and directed *edges* $e : a \rightarrow b \in G_E$ from a *source* node $s_G(e) = a$ to a *target* node $t_G(e) = b$. Nodes and edges of a graph are labeled by attributes. Nodes represent RBAC entities such as users, roles, permissions, operations, and objects (cf. definition of RBAC models [FSG+01]). The edges of the graph represent the RBAC relations such as UA, PA, and RH.

A *graph rule* $r : L \rightarrow R$ consists of a graph L (left-hand side), a graph R (right-hand side) and partial mappings $r_N : L_N \rightarrow R_N$ and $r_E : L_E \rightarrow R_E$ which preserve the graph structure, i.e., $s_R(r_E(e)) = r_N(s_G(e))$ and $t_R(r_E(e)) = r_N(t_G(e))$ for all edges $e \in dom(r_E)$. The graph L describes the elements a graph must contain for r to be applicable. The mappings are undefined on nodes/edges that are intended to be deleted, defined on nodes/edges that are intended to be preserved. Nodes and edges of R without a pre-image are newly created. Note that the actual deletions/ additions are performed on the graphs to which the rule is applied. Figure 1 shows the graph rules $assignUser/removeUser$, $assignRight/removeRight$ and $newUser/deleteUser$.

As a consequence, the administrative RBAC functions, which change the RBAC configuration, can be represented by graph rules in a natural way. In addition, we can define certain conditions for each graph rule. In order to apply those rules, the rule conditions must be satisfied. Otherwise, the rule cannot be performed. If we define now authorization constraints (such as SoD, prerequisite roles, cardinality constraints) as the conditions for an administrative RBAC function (which is represented by a graph rule as mentioned above), the function in question cannot be performed if any relevant authorization constraint is violated.

As a result, we can produce only RBAC configuration states which are consistent to the autho-

rization constraints (i.e., to the defined RBAC policy) as shown by Koch et al. [KMPP02].

2.3 DTAC model, Entity-Relationship model

The DTAC (dynamically typed access control) model regards concepts as basic entities for authorization schemata. The corresponding formalism (see [TP01]) uses a notation derived from E-R diagrams and category theory to represent authorization schemata. A concept is symbolized by a *concept node* (depicted as a rectangular box). Concept nodes represent type classes or aggregations of instances of some other concept node, thus an *instance* of a concept node is a *type* or an *aggregate*. Type classes are homogeneous collections of system specific types, whereas aggregation defines a membership relationship without further conditions. Inheritance can only happen between types that belong to the same type class. Thus inheritance can only occur within a single type class. In order to keep the DTAC model well defined, cyclic constructions of either inheritance or aggregation are forbidden, which also eliminates mutual cycles. An edge between two concept nodes indicates the existence of a relationship. Edges in DTAC schemata are represented in terms of standard UML edges (cp. Figure 1 in [TP01]). A ternary relation is drawn as a three way intersection with a small solid dot.
Instances of concept nodes are depicted by circles. There are two types of edges in DTAC instance graphs: *assignment* edges which are solid arrows and *constraint* edges which are dashed (cp. Figure 14 in [TP01]).

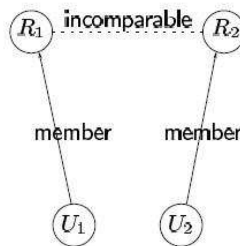

Figure 2: Only separate users are allowed to be members of exclusive roles.

A constraint between a pair of aggregates will be marked with a small circle at both ends of the constraint edge. Assignment edges must go from an instance of a concept node to another instance of a concept node. These concept nodes must be linked by a relationship in the underlying DTAC schema. Constraint edges may go between any pair of instances belonging to different concept nodes. Cardinality constraints limit the number of allowed assignment relationships between the instances indicated by the constraint. Constraints on aggregates apply across all members of the aggregate. Figure 2, which is a copy of Figure 1 in [TJ00], presents a constraint between two exclusive roles.
The dynamic aspects of access control models are reflected by triggers. Triggers are specified as graph transformations which consist of before and after configurations and may optionally include constraints. In this manner changes of a DTAC graph that represent the flow of jobs or tasks in a workflow may be expressed.
Overall, the DTAC model can be regarded as an approach to model access control by algebraic generalizations. It is probably not meant for reasoning about single objects or users since the

instances of concept nodes are aggregates or types. As the Tidswell et al. point out, they believe introducing mechanisms for abstraction which eliminate inappropriate details will increase the robustness of the security specifications. Furthermore, the DTAC model separates the structure from the constraint specification. Tidswell et al. see the DTAC model as a solid mathematical basis for future work. Thus the DTAC model may be a good starting point for policy design since Tidswell et al. intended to define a fully general constraint specification language.
On the other hand, the DTAC formalism seems to be not so adequate for modeling temporal properties and dynamic authorization constraints as (first-order) LTL, which is a disadvantage. Moreover there seems to be no tool support for the DTAC model currently. In the opinion of the authors this might possibly be overcome by the application of the USE tool (see subsection 3.5) or Alcoa (see subsection 3.3) and/or a graph transformation tool like AGG (see subsection 3.4). Clearly, it remains still to be analyzed whether Alloy is an appropriate formalism for the specification of the DTAC model.
Compared to the DTAC model entity-relationship diagrams are a less general formalism, but also mathematically solid. An entity in the entity-relationship model can be anything that can be distinctly identified. This facilitates reasoning about single objects or users in contrast to the DTAC model. Relationships are (ordered) tuples of entities representing an association among the entities in question. Each position in the ordered tuple may correspond to a certain role defining the function of the entity on this position w.r.t. the considered relationship. An attribute is a function mapping from an entity set or relationship set into a value set or a Cartesian product of value sets. The values contain the information about an entity or a relationship, which can be obtained by observation or measurement. According to Chen the primary purpose of entity-relationship diagrams is to serve as a tool for database design. For further details on the entity-relationship model the reader is referred to [Che76].
As explained in [Che76] the relationship sets hold semantic information about data dependencies. The entity-relationship model allows to define rules enforcing data consistency in case of updating, inserting and deleting the database in question since it facilitates understanding and specifying constraints for maintaining data integrity. If the contents of the database is the current state of the RBAC configuration (with users, objects, RBAC roles, etc. as entities), the entity-relationship model thus may help to define the rules for updating, inserting and deleting such that the resulting state transitions are consistent with the given RBAC security policy (cf. [KMPP02]). Under the tacit assumption of adequate tool support, the entity-relationship model may hence serve as a basis for the construction and implementation of an authorization engine which decides, whether an access request should be granted or not.

2.4 UML, OCL, TOCL

An approach to specify authorization constraints for role-based access control is based on the *unified modeling language (UML)*. UML is a general-purpose modeling language which can represent the sets, relations, and functions of the RBAC model [SA00]. Using the *object constraint language (OCL)* that is part of the UML standard, it is possible to specify authorization constraints in OCL for the UML model of RBAC. OCL has the same language constructs as first-order logic including `Forall` and `Exists` which are, however, not applicable to infinite sets [HDF02]. Hence, we expect the expressiveness of OCL authorization constraints to be similar or even the same as the one of RCL 2000 [Ahn99], which is equivalent to a restricted version of first-order logic. For example, Ahn and Shin [AS00] state a variety of authorization

constraints using OCL including static and dynamic separation of duty constraints, prerequisite constraints, and cardinality constraints. There is an extension of OCL with temporal logic called *TOCL* [ZG02] that augments OCL by the temporal capabilities of a finite linear temporal logic. As a consequence, TOCL is able to express history-based authorization constraints that depend on past and future conditions.

In the rest of this subsection, we deal with aspects of specifying authorization policies using this approach with regard to policy validation.

To have a precise and unambiguous understanding of the meaning of the expressions of a language is almost a prerequisite for validation. Otherwise it is not possible to find out precisely whether a given statement of the language really expresses what it is supposed to. For this reason OCL and TOCL have formal semantics [Ric02, ZG02, RG02] and there is work going on to provide a unified formal semantics for UML as well [Sof]. It should be noted that according to Kleppe and Warmer it is important to ensure that the formal semantics for the various modeling concepts of UML and OCL are based on one unified semantic domain [KW04].

As mentioned in subsection 1.2, there are two different approaches to validating authorization policies: configuration-based validation and deduction-based validation. As OCL does not provide any inference rules, on the one hand, deduction-based validation is not applicable for authorization policies specified in OCL. It would be necessary to translate the policy to an appropriate logical calculus first. This is a minus for OCL. On the other hand, there are several configuration-based validation approaches for OCL [RG02]. Since OCL expressions are always computable (because of OCL constructs being applicable only to finite sets), OCL is decidable and thus it is a good foundation for full-automatic configuration-based validity checks at runtime [HDF02]. One of the configuration-based validation approaches is based on the USE tool [RG00], which is a validation tool for UML models and OCL constraints. In particular, there has been research on how to use this tool for authorization policy validation [SAGM05]. However, the USE tool is not able to deal with TOCL. Subsection 3.5 gives more information about the USE tool.

Finally, considering the fact that OCL provides (configuration-based) validation capabilities, its wide-spread and even commercial use is a big plus. Apparently, this wide-spread use is due to the fact that OCL, although being to some extent related to a mathematical foundation, does not look like a mathematical formalism. It does not have any mathematical symbols [WK03, p.17].

3 Supporting tools for Validation

3.1 The Theorem Prover Isabelle

Isabelle [NPW02] is a generic proof assistant based on the functional programming language ML. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. The main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols.
Compared with similar tools, Isabelle's distinguishing feature is its flexibility. Most proof assistants are built around a single formal calculus, typically higher-order logic. Isabelle has the

capacity to accept a variety of formal calculi. The distributed version supports higher-order logic but also axiomatic set theory and several other formalisms. Formal calculi and/or mathematical theorems are encoded in theory files (suffix `.thy`) that can be invoked to Isabelle by the command `use_thy`. 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.
Isabelle can be viewed from two main perspectives:

- On the one hand it may serve as a generic framework for rapid prototyping of deductive systems.
- On the other hand, major existing logics like Isabelle/HOL provide a theorem proving environment ready to use for sizeable applications.

Isabelle/HOL is currently the best developed object logic, including an extensive library of (concrete) mathematics, and various packages for advanced definitional concepts like (co-)inductive sets and types, well-founded recursion, etc. The distribution also includes some large applications, for example correctness proofs of cryptographic protocols (HOL/Auth) or communication protocols (HOLCF/IOA).
The authors have embedded the first-order LTL in Isabelle/HOL and verified some properties of an RBAC policy as described in 1.2.1 and published in [DBTS04]. Unlike model checkers, Isabelle/HOL in general cannot run into complexity problems since Isabelle as a theorem prover just has to check a given proof, which mostly implies a reasonable upper bound for the complexity [4].
Of course, this advantage of Isabelle/HOL compared to model checkers is not for free because the human user has to provide correct proofs for the given theorems. This may be time-consuming and impose considerable effort on the policy designer, but this labor can result in theorems and lemmata that are much more general than the results of model checkers and might thus be applied to a variety of possible scenarios. Furthermore, proven theorems and lemmata can arbitrarily be reused in following proofs. By contrast, such a reuse of previous results is generally not possible for model checkers.

3.2 The Model Checker NuSMV

Various model checking tools exist. For a reference see [CGP99]. In this section, we discuss the NuSMV model checker which will be later employed for the verification of workflow SoD properties. The NuSMV [CCG+02] is a symbolic model checker, which is an extension of McMillan's SMV system [K.L92]. Beyond SMV's BDD-based model checking NuSMV now supports also model checking techniques based upon propositional satisfiability.
The FSM can be specified by an intuitive input language. Since it is intended to describe FSMs, the only data types are finite ones, namely Booleans, scalars, and fixed arrays. In addition, reusable components can be specified by modules. The primary purpose of NuSMV's input language is to describe the transition relation of the FSM in question. For this purpose, `next` expressions can be used. For example, if we have specified `next(b):=1;` for a Boolean state variable `b`, this means that in the following state `b` is true. Moreover, with the help of the `init` function, we can also define initial values for state variables (remember that an FSM has a set of initial states).

[4] Some incorrect proofs can cause an infinite loop with Isabelle's term rewriting, but at least for correct proofs there is usually a reasonable upper bound for complexity.

3.2.1 Model Checking RBAC Configurations for Workflows

As indicated earlier (cf. Section 2.1, we often must deal with *dynamic* security properties in the context of workflows. Examples are the various kinds of dynamic SoD policies as given by Simon and Zurko [SZ97]. In summary, our model checking-based approach to validation of RBAC configurations for workflows works as follows: The RBAC configuration, the workflow itself and the delegation and revocation steps are specified by means of an FSM, and then the SoD properties to be checked are specified in propositional LTL. The model checker then takes care of checking the security properties in question. Followingly, we present two examples of specifications of separation of duty properties in propositional LTL.

Simple Dynamic SoD (SDSoD) A principal may be a member of any two exclusive roles but must not activate them at the same time:

```
!(activate_u_clerkpreproc & activate_u_clerkpostproc).
```

There is a loophole with this property: The exclusive roles could be activated one after another. Hence, a better version for SDSoD would be, for example:

```
(activate_u_clerkpreproc -> !  F activate_u_clerkpostproc).
```

Operational Dynamic Separation of Duties (OpDSoD) A principal may be a member of some exclusive roles as long as the set of operations acquired over these roles does not cover an entire workflow.

Here we need to relax 'entire workflow' to steps 1.-9. of the banking workflow given in [SLS06] only. This can theoretically be done by the two roles `ClerkPreProcessor` and `ClerkPostProcessor` only (if we do not exceed the 100k thresholds). This would mean to check for two things:

1. they are not assigned over two roles at any state,
2. they have not been delegated and revoked one after the other over some states such that never at any state all authorizations cover 1.-9.

Once again, the second variant - which is stronger than the first one - prevents a principal from circumventing OpDSoD by delegation and revocation. In propositional LTL, this second variant can now be expressed as follows:

```
!(F auth_u_update_customerdata & F auth_u_query_customerdata
& F auth_u_prepare_ratingreport & F auth_u_release_ratingreport
& F auth_u_post_ratingreport & F auth_u_query_ratingreport
& F auth_u_queryavailproducts_productbundle &
F auth_u_update_productbundle & F auth_u_commit_productbundle);
```

Note that we introduced here further state variables indicating that principal u is authorized to execute the operations in question such as update, query, or prepare.

3.2.2 Analysis of the Interplay between SoD Properties and Delegation

Maintaining an operational dynamic SoD property requires an additional piece of information: The underlying workflow model within a specific instance of which objects are accessed. We then need to compare the set of required authorizations for a workflow with the critical set of authorizations computed from a principal's exclusive roles. In addition, we need to check for any delegation and revocation activities within the duration of a workflow and whether these resulted in the acquisition of the critical authorization set.
During our analysis we have encountered a scenario by means of NuSMV where the aforementioned naive version of OpDSoD (cf. Section 3.2.1) can be circumvented whereas the second variant of OpDSoD cannot. In this scenario principal `u1` delegates the access rights of his `Clerk PreProcessor` role to `u2` one after the other. After `u2` has exercised this access right, `u1` immediately revokes it. We further assume that `u2` has the role `Clerk PostProcessor`. If we did not exceed the 100k limit, then the supervisor is not involved, and `u2` could execute all operations covering the steps 1.-9. of the banking workflow. This way, OpDSoD could be violated. In our test run, the first property is evaluated to true by NuSMV whereas the second is false. In fact, the first constraint is satisfied at every time step: `u2` never has all the authorizations of the critical set at a point of time. NuSMV also gives a counterexample (trace) in case of the second (stronger) property.

3.2.3 Inter-Instance Analysis of Workflows

Up to now, we have only dealt with an analysis of a **single workflow instance** by checking whether the RBAC configuration adheres to certain security properties. By means of NuSMV, we can carry out an inter-instance analysis, too. For example, a business object such as `credit_for_A` may be accessed in different instances of a loan origination workflow [5]. This way, it might be possible to circumvent certain SoD rules by accessing the business object in both workflow instances. In particular, we can now differentiate between two cases of an inter-instance analysis:

- inter-instance analysis of workflow instances of the **same schema/model** (e.g., of schema loan origination workflow),
- inter-instance analysis of workflow instances of different schemata/models (e.g., an instance of a shipping of goods workflow and an instance of an order-to-stock workflow).

For the inter-instance analysis, we use the `process` statement provided by NuSMV. Processes are used to model interleaving concurrency. A process is a NuSMV module which is instantiated using the keyword `process`.

A NuSMV module declaration is an encapsulated collection of declarations and specifications. A module declaration also opens a new identifier scope. Once defined, a module can be reused as many times as necessary. Modules are used in such a way that each instance of a module refers to different data structures. Furthermore, a module can be called with parameters. The actual parameters can be regarded as data of a higher level scope which are handed over to the module in question.

[5] The following discussion is based upon [SLS06], where the reader can find the details.

The program now executes a step by non-deterministically selecting a process, then executing all of the assignment statements (such as `next` statements) in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. The LTL specifications such as object-based dynamic SoD can then be checked against the concurrent system.

3.2.4 Authorization Engine for RBAC Policies

One can employ the model checking approach for implementing an authorization engine which can enforce the dynamic and history-based policies such as object-based dynamic SoD. The basic idea of such an authorization engine is only sketched subsequently.
Whenever a user/principal performs a task instance with the help of a workflow engine, it must be checked if the access is to be granted. For this decision, we must take into consideration which tasks instances have already performed, i.e., we need the access history. For example, if a user has executed the `update product bundle` operation, then `commit product bundle` must not be executed by the same user. The model checker can now be called by the authorization engine (which acts a policy decision point) to decide whether the access is granted according to the access history. To sum up, we utilize NuSMV's ability to check temporal properties at this point.

3.3 Alcoa

Alcoa is a tool for analyzing object models which are specified in the input language Alloy [6]. Alloy (cp. figure 3) is basically a first-order logic with a relational calculus offering a syntax for structuring specifications in the logic. Every value in the Alloy logic is a relation and all relations in Alloy are first-order, i.e. a relation cannot contain other relations and no sets of sets. The notation of Alloy has been inspired by Z.
Alloy supports the description of systems whose state involves complex relational structures. Alloy is a declarative language that enables one to build a model by layering properties using conjunction. This allows the construction of partial models, in which constraints describe how state components are related to one another, without explicit rules for how each component is updated.

Alcoa essentially translates constraints specified in Alloy to boolean formulas and then applies state-of-the-art SAT solvers, which allows to analyze billions of states in seconds. Since Alloy is not a decidable language, Alcoa cannot provide a sound and complete analysis. Instead it conducts a search within a finite scope chosen by the user that bounds the number of elements in each primitive type. Thus Alcoa is fundamentally a compiler which translates the problem to be analyzed in a (usually huge) boolean formula. This formula is handed to a SAT solver, and the solution (if a such exists) is translated back by Alcoa in the language of the model (which has been specified in Alloy). Otherwise the user receives the message that no solution was found in the given scope.
Alcoa helps to address two principal risks of declarative modeling:

[6]The following description is mainly based upon [JSS00].

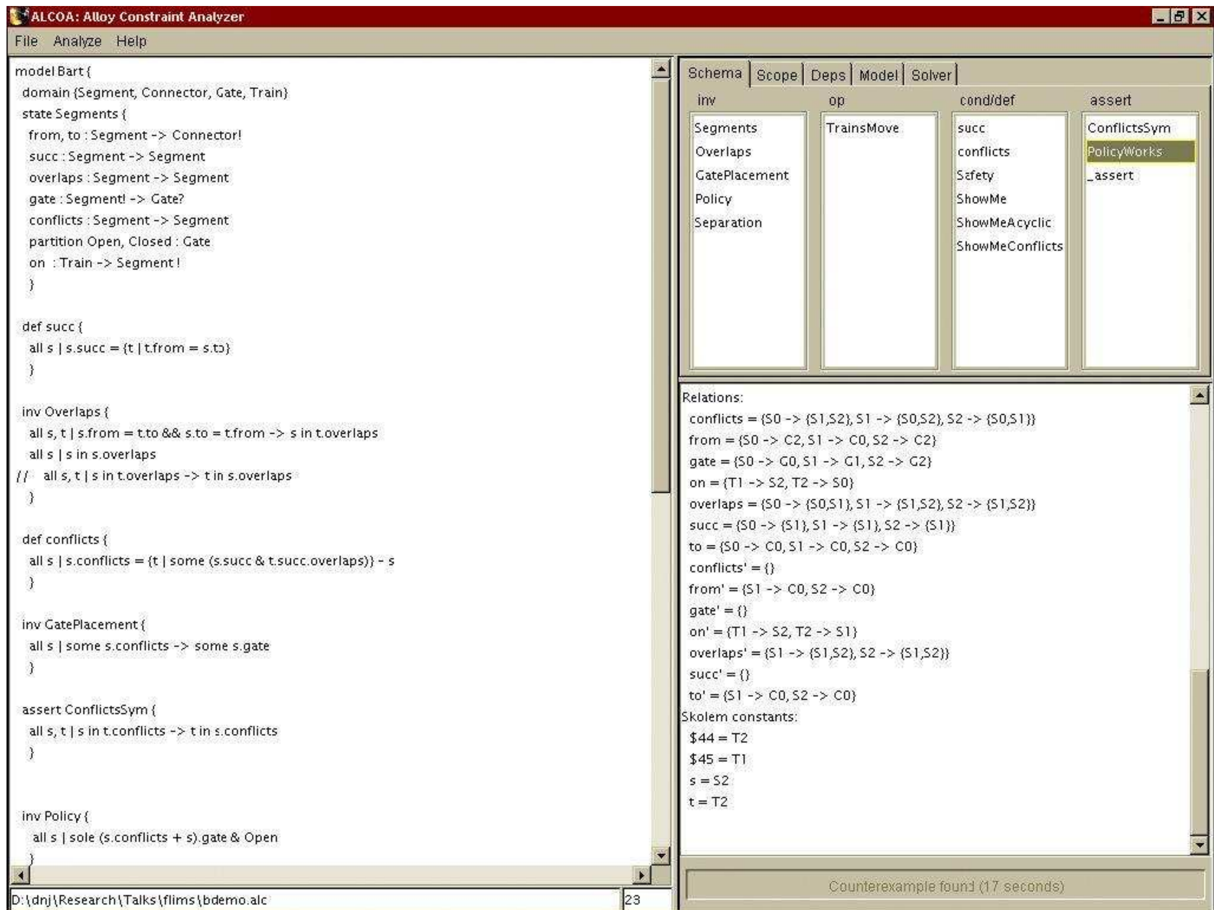

Figure 3: Screenshot of Alcoa (copy of Figure 1 in [JSS00]).

- wrong assertions
- contradictory constraints

For the first risk Alcoa offers the possibility to check an assertion, i.e. a theorem that certain consequences follow from the given constraints. Alcoa searches now for counterexamples within the given scope. If it finds one the assertion is not valid. On the other hand, if Alcoa finds no counterexample, the theorem can still be not valid. In contrast to that, a correct theorem prover can definitely confirm the validity of a theorem, if it is given a sound proof.
For the second risk Alcoa can exercise invariants or operations. If it finds an example, the user can be assured of the consistency of the constraints in question. If no such example is found within the given scope, the given constraints could still be consistent. Thus altogether Alcoa may be helpful for automated validation, but it is not suited for deduction-based validation. Therefore it is a lot closer to model checkers w.r.t its abilities than to theorem provers. Unlike model checkers it does not provide the possibility to investigate elaborate temporal properties. In contrast to model checkers, Alcoa addresses the complexity that arises from relational state structure. Thus Alcoa is quite similar to the USE tool.
Similar to model checking, the complexity is also for Alcoa an issue of grave importance. As the reader may know, the 3-SAT problem is NP-complete and its complexity is $O(2^n)$ in the worst case. Thus depending on the problem it may become a question of time whether it makes sense to employ Alcoa for the solution (cp. [Sch03]). This is a clear disadvantage w.r.t. theo-

rem provers. The reader may find detailed information on the application of Alcoa w.r.t. SoD properties in [Sch03].

3.4 Validation with Graph Transformation Tools

Let there be a set of organizational rules (authorization constraints) such as separation of duty rules, cardinality constraints and prerequisite roles. How can one check whether a given RBAC configuration state remains consistent to this RBAC policy when administrative RBAC functions such as assign user to role, add user, deassign user are carried out?

To solve this problem we can use graph transformations [Roz97]. The basic idea of the graph-based approach to validation of RBAC policies is as follows according to Koch et al. [KMPP02]: The current RBAC configuration state respectively protection state is represented by a graph (cf. Section 2.2). The nodes represent RBAC users, roles, operations, objects and sessions (cf. definition of RBAC models [FSG+01]). The edges of the graph represent the RBAC relations such as UA, PA, and RH. Assuming that the current RBAC configuration state is consistent with the given RBAC policy or more precisely authorization constraints, then we can make sure that graph transformations preserve this consistency by making the authorization constraints post-conditions for any graph transformation step in question we want to apply. Thus it is assured that only graph transformations which are consistent with the given RBAC policy can be applied to the current RBAC configuration state or protection state. So we are assured that any modification of the protection state will preserve the consistency.

In order to implement the aforementioned theoretical graph-based approach, we can employ general-purpose graph transformation tools such as AGG [TER99]. In Figure 4, a screenshot of AGG is shown. Due to the fact the AGG tool has a built-in constraint validation mechanism one can also use the AGG tool in order to detect conflicting and missing constraints as can be done by means of the USE system (cf. 3.5). This validation mechanism allows for checking the current graph state against defined (authorization) constraints. The graph-based approach and the UML-/OCL-approach are similar this way (it has still to be investigated which approach is more efficient).

In addition, AGG supports also a Java API (similar to the USE system) and hence it is easily possible to build a policy decision point for RBAC policies containing authorization constraints (authorization engine). Similar to the USE approach, however, history-based SoD constraints as required in the context of workflows are not supported. Here, the model checking and the deduction-based formal validation approaches are better suited (cf. Section 3.1). Moreover, the graph-based approach does not allow for a **formal** validation of RBAC policies. Strictly speaking, only the consistency of the current RBAC configuration against the defined RBAC policy can be checked.

3.5 Specification and Validation with UML and OCL

Problem: Given is a set of organizational rules (authorization constraints) such as separation of duty rules, cardinality constraints and prerequisite roles. Can we check whether we have defined redundant rules or to put it the other way round, are certain rules missing which the policy designer has in mind? Furthermore, we would like to check if the set of rules is consistent.

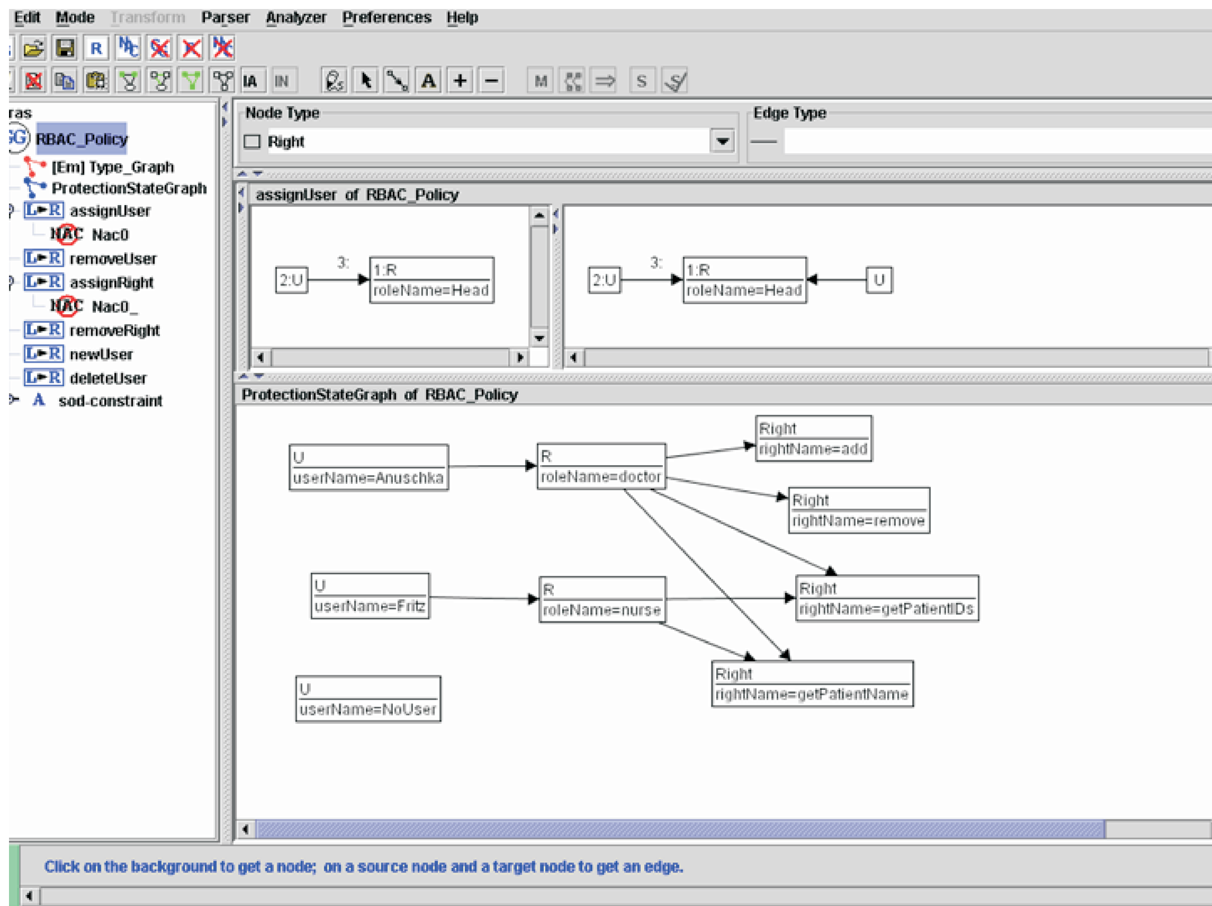

Figure 4: The graph transformation tool AGG.

On the other hand, we would like to use a light-weight validation approach for this purpose employing formalisms/modeling languages which are widely used such as the Unified Modeling Language.

Due to the fact that theorem provers are still tools for experts, the formal deduction-based validation approach is strictly speaking not necessary in applications with moderate security requirements. For this reason, we present a more practical approach to policy specification and analysis. Clearly, we could 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 validation step.

For the practical specification and validation, we employ UML and the Object Constraint Language (OCL)[7] [WK03]. As demonstrated in [AS01, RLFK04], 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.

Subsequently, we briefly introduce the USE system and thereafter describe several use cases in the context of validation of RBAC policies.

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

3.5.1 The USE System

OCL is a light-weight formalism, which can help in specifying RBAC policies. We now demonstrate how the USE tool [Ric02] can be employed for the validation of RBAC policies formulated in UML/OCL. In the following, we explain the functionality of USE.

USE allows the software modeler to validate UML and OCL descriptions and is the only OCL tool allowing interactive monitoring of OCL invariants and pre- and postconditions, and the automatic generation of non-trivial system states. These *system states* or *system snapshots* consist of the current objects and links between those objects adhering to the UML model in question.

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

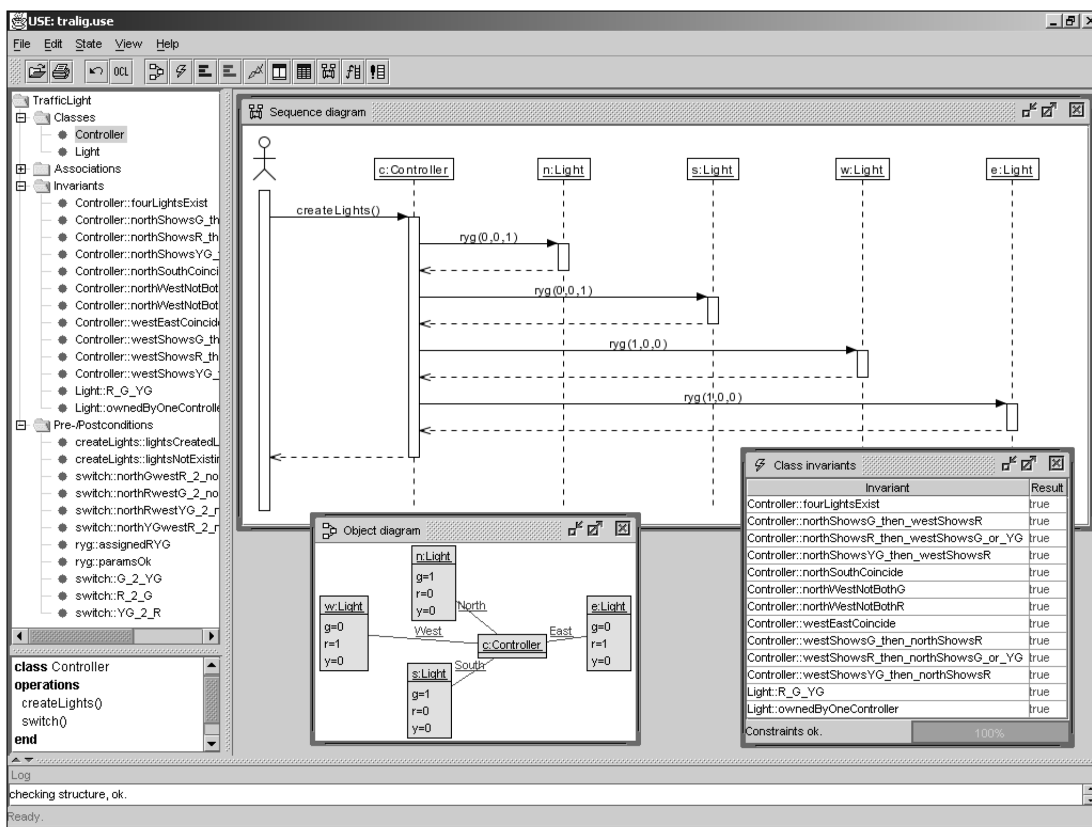

Figure 5: USE screenshot.

The USE tool expects as an input a textual description of a model and its OCL constraints (for an example of such a description refer to Fig. 7). After syntax checks, the model can be displayed by the graphical user interface provided by USE. In particular, USE makes available a project

browser which displays all the classes, associations, invariants, and pre- and post-conditions of the current model.

Fig. 5 shows a USE screenshot with an example. On the left, we see the project browser displaying the classes, associations, invariants, and operation pre- and post-conditions. In a detail window below, the selected class is pictured with all details. On the right, we identify a sequence diagram presenting the operations which lead to the current system state given in the object diagram window below. The evaluation of the invariants in this system state is pictured in the class invariant window to the right of the object diagram window. The developer gets feedback from USE about the validity of the invariants in the invariant window and the validity of the pre- and post-conditions in the sequence diagram window.

The USE tool can be employed in various ways in the context of RBAC policies (cf. Fig. 6). Specifically, it can be employed for the specification (cf. Fig. 7) 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 3.5.3. The last use case is testing concrete RBAC configurations, i.e., after deployment of the policy [SAGM05].

3.5.2 Validation of RBAC Policies

We now briefly describe how to employ the USE system (UML-based Specification Environment) [Ric02] to validate 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 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) as mentioned above.

Validation with the USE system could 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 a forbidden 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 RBAC configurations cannot be established. A typical example of two contradictory rules is a prerequisite role constraint between the roles *r1* and *r2*. This rule states that a user may only be assigned to *r2* if she is at the same time assigned to *r1*. However, if we now define a simple static SoD rule between the role *r1* and *r2*, then we obviously cannot satisfy both rules at the time. Hence, both constraints are contradictory.

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 the theorem proving approach described in previous sections.

3.5.3 Authorization Engine

Beyond specification and validation, 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. We can also employ the USE system, which is a validation tool for software models, to implement an authorization engine. For this purpose, we can use the Java API made available by USE [Ric02]. This engine serves as a key component for enforcing various types of authorization constraints. This way, RBAC policies for different organizations can be implemented.

The basic idea of the constraint checking mechanism of the authorization engine 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.

3.5.4 Testing a Given RBAC Configuration with USE

Beyond the validation of RBAC policies, USE can be employed for testing an RBAC configuration *after* the constraints have been deployed. However, observe that we consider here a *predefined* RBAC configuration of users, roles, etc. which corresponds to a real-world RBAC configuration of an organization.

Testing an RBAC configuration may be mandatory in several situations. For example, in some domains (e.g., healthcare) strict data protection laws must be fulfilled such as the European Directive 95/46/EC [EU95]. In order to assess the current RBAC configuration defined for security-relevant applications, often some external review is required, e.g., from an government agency responsible for data protection as established in Germany. What is often missing is a tool that supports an external reviewer in checking a concrete RBAC configuration of an organization against certain properties such as data protection rules. In addition, the ability to test RBAC configurations may also be helpful for administrators in order to check if a security policy has been implemented correctly.

USE can now be employed as an ad hoc query tool to check certain properties of the current RBAC configuration such as:

- there is no common user of mutually exclusive roles
- only clinicians of a patient's current ward may have access to the patient's electronic patient record[8]

[8] We assume here that there is a further attribute "ward" for certain roles and for users.

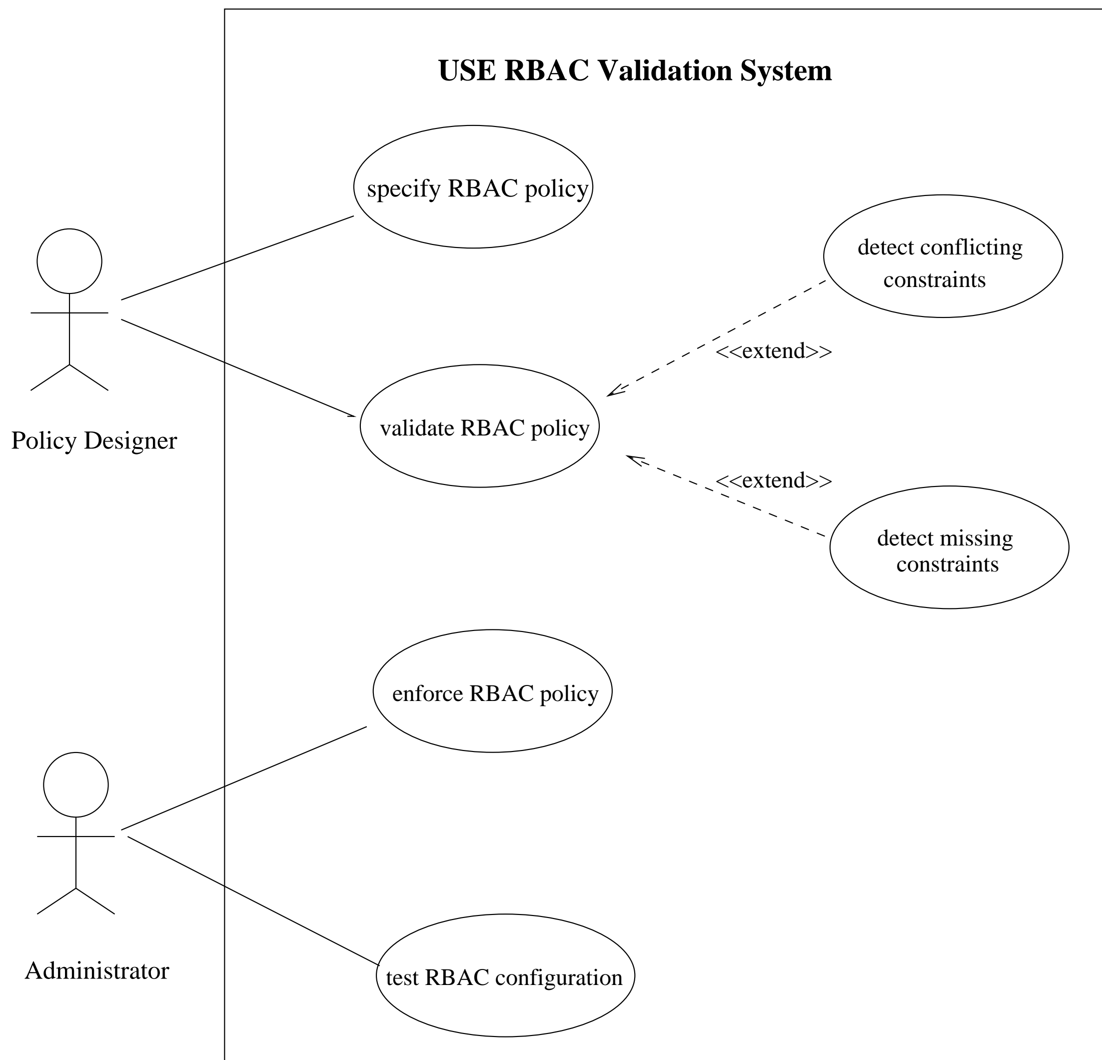

Figure 6: Use cases for the RBAC USE system.

4 Assessment of the considered formalisms and tools

Evaluating the described formalisms and tools w.r.t. validation results in the following final conclusions.

4.1 Formalisms

All considered formalisms have a mathematically defined semantics, which was named as a necessary condition for validation in workpackage 2.1 (cp. subsection 6.4).

```
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))
```

Figure 7: USE specification of an RBAC policy.

4.1.1 (First-order) LTL

This is the only formalism providing inference rules and therefore meeting the requirements for deduction-based validation. From the viewpoint of mathematics, it may be considered the strongest formalism since it offers a temporal structure and all the capabilities of a first-order logic, which is a sufficient basis for many parts of mathematics (e. g. Zermelo-Fraenkel set theory including axiom of choice). Therefore it allows to express very general concepts such as various kinds of delegation. On the other hand, introducing constants one can also describe conditions and/or restrictions for single users or objects. So first-order LTL is a formalism with a high flexibility.
Propositional LTL is in contrast to first-order LTL decidable. That is to say, there is a terminating deterministic algorithm for deciding whether a given formula of propositional LTL is satisfiable because propositional LTL has no predicates and no quantifiers. Propositional LTL allows to specify regular RBAC configurations respectively Finite State Machines (FSM). Thus it is a good formalism for automated validation.
First-order LTL has been encoded in the theorem prover Isabelle (cp. [DBTS04]). For propositional LTL the model checker NuSMV (cp. [SLS06]) is an adequate tool.

4.1.2 Graphs

The graph-based formalism introduced by Koch et al. represents the current state of the RBAC configuration as a graph. Hence, the dynamic changes of this state can be expressed by graph transformations. It is now possible to modify the rules of an arbitrary graph transformation by a terminating deterministic algorithm such that they are consistent with a given RBAC security policy or more precisely a finite set of authorization constraints. The modified graph transformations will now preserve consistency of RBAC configuration states, i.e., consistent states can never be transformed to inconsistent states by the modified graph transformations (see [KMPP02]). In contrast to (first-order) LTL, this formalism does not have any inference rules. Furthermore, it is not possible to reason about arbitrary finite or even infinite sets of users, objects etc. Finally, there is no explicit representation of temporal structures. The described formalism may thus serve as a basis for an authorization engine [9], but is not comparable to first-order LTL w.r.t. flexibility and significance.

4.1.3 DTAC model, Entity-Relationship model

The DTAC model is a very general algebraic approach representing *concepts* as graph nodes whose instances are types or aggregates (see [TP01]). This formalism is able to express very general authorization schemata, which can then be specialized to DTAC instance graphs. The edges between graph nodes represent relations between concepts, between instances of graph nodes one may additionally have edges representing constraints. The dynamic aspects of access control are reflected by graph transformations of DTAC instance graphs. As already mentioned (cf. subsection 2.3), this formalism is well-suited to handling general concepts, but does not seem to be apt for expressing constraints for single users or objects. As Tidswell et al. annotate, the constraint framework might possibly be too general to impose sufficient restrictions on users since authorization constraints such as object-based dynamic separation of duty refer to entities like single users or objects .
Similar to the graph-based formalism discussed previously, there are no explicit representation of temporal structures and no inference rules. So the DTAC model is also not comparable to first-order LTL w.r.t. flexibility and significance, but it may be a good starting point for policy design (if adequate tool support can be found).
The entity-relationship model is less general than the DTAC model since by definition entities can always be distinctly identified as single users or objects. Entity-relationship diagrams are primarily a tool for database design. Relationships and attributes are an explicit representation for important semantic information about the database in question as for example data dependencies. The entity-relationship model allows to define rules for updating, inserting and deleting of data such that the integrity of the database is preserved. If the considered database is the current state of an RBAC configuration the entity-relationship model should help to define the rules for deleting, inserting and updating consistently with a given RBAC security policy, i.e. no consistent state of an RBAC configuration could become inconsistent by deleting, inserting and updating. So the entity-relationship model might be a basis for an authorization engine similar to graphs (see subsection 4.1.2). In that case, a database management system might be a good candidate for a supporting tool.

[9] This has been demonstrated in a master thesis using the AGG tool, which was also used in [KMPP02]

Analogously to the DTAC model the entity-relationship model offers neither inference rules nor an explicit representation of temporal structures. Thus it is definitely less powerful than (first-order) LTL.

4.1.4 UML,OCL,TOCL

As explained in subsection 2.4, the general-purpose modeling language UML can be used to specify the RBAC model. The reader may notice the similarity of UML with the DTAC model (cp. subsection 2.3). The Object Constraint Language of the UML standard (OCL) allows then to express authorization constraints for the RBAC model specified in UML. OCL has the same quantifier symbols as first-order logic, but they are not applicable to infinite sets. TOCL augments OCL by the temporal capabilities of a finite linear temporal logic. As mentioned in subsection 2.4, there is a formal semantics for OCL and TOCL. Furthermore, the USE tool allows to evaluate OCL constraints for a given UML model such as the RBAC model (specifically authorization constraints). For TOCL there is not yet a supporting tool.
On the other hand, OCL is not semantically equivalent to a first-order logic since it is decidable and does not allow quantification over infinite sets. Thus TOCL can also not be equivalent to a first-order LTL. Finally, there are no inference rules for OCL and TOCL.

4.2 Tools

The regarded tools support one or more of the considered formalisms. One essential objective of the presented tools is minimizing the probability of human mistakes in the application of the considered formalisms.

4.2.1 Isabelle

As explained in subsection 3.1, the theorem prover Isabelle is a generic proof assistant providing a high flexibility, i.e. it has the capacity to accept a variety of formal calculi. It may serve as a generic framework for rapid prototyping of deductive systems. Thus it is quite appropriate to formalize large parts of mathematics (including classical higher order logic, Zermelo-Fraenkel set theory and many other theories). Since Isabelle facilitates deduction, it is a good tool for deduction-based validation. Furthermore, many mathematical theories and extensions exist and can be downloaded from the Isabelle Library (see [PN]). Isabelle is not dependent on a finite scope of the given problem such as model checkers, graph transformation tools, the USE tool or Alcoa. Accessorily, deduction-based validation can prove a lot more general theorems than configuration-based validation. Finally, complexity is not an issue for Isabelle in general because it checks given proofs for correctness (cp. subsection 3.1), which is much less time consuming than finding correct proofs. But a proven theorem or lemma can be reused arbitrarily in further proofs.
On the other hand, finding helpful theorems and supplying correct proofs for them may require

time and good mathematical skills. Thus the human user has to go the time and effort of finding correct proofs for the theorems in question. As a general rule, deduction-based validation should not be used, if the same result can be achieved by configuration-based validation.
Overall, Isabelle/HOL together with first-order LTL as encoded formalism can be regarded as a powerful tool for deduction-based validation in ORKA. It can be quite helpful w.r.t. proving security properties of RBAC policies. Combining first-order LTL encoded in Isabelle/HOL with a model checker like NuSMV (see subsection 3.2) for checking the consistency of the given RBAC policy seems to meet all essential requirements for validation in ORKA.

4.2.2 NuSMV

The symbolic model checker NuSMV is an extension of of McMillan's SMV system [K.L92] (see subsection 3.2). It supports model checking techniques based upon propositional satisfiability. It can be used for a model checking-based approach to validation of security properties of workflows (see subsection 3.2.1). Unlike Isabelle, NuSMV is restricted to decidable problems and can essentially run without human help. On the other hand, deduction is not possible with NuSMV. In theory, model checkers like NuSMV could be used for arbitrary security requirements in order to assure that no violation occurs. This could be done by checking the RBAC configuration state resulting of the state transition associated with an arbitrary access request for consistency with all given security requirements (assuming the consistency of the initial state with all security requirements) and granting only those requests being consistent with all security requirements. The described procedure would not depend on human intervention, but unfortunately in contrast to Isabelle complexity is an issue for NuSMV and other model checkers. Since any access request would require a new evaluation of the security requirements, the IT system might become much too slow in practice. But as pointed out in subsection 3.2.3, NuSMV may be used to avoid inter-instance conflicts of workflow instances. Furthermore, NuSMV may be used to prove the consistency of a given RBAC policy (cp. subsections 1.2, 2.1.1 and 4.2.1).

4.2.3 Alcoa

As pointed out in subsection 3.3, the constraint analyzer Alcoa is a tool for analyzing object models specified in the input language Alloy. Although Alloy is basically a first-order logic with a relational calculus, Alcoa does not support deduction. It translates the problem to be analyzed in a (usually huge) boolean formula and passes this formula on to a state-of-the-art SAT solver. Thus a finite scope of the problem is required. Alcoa is able to analyze a multitude of states within seconds. As mentioned in subsection 3.3, Alcoa helps to discover wrong assertions and contradictory constraints. In contrast to model checkers, Alcoa does not offer the possibility to investigate elaborate temporal properties. One can conclude that Alcoa is rather similar to the USE tool, although possibly less dependent on human intervention. Thus Alcoa is not well-suited for workflows and authorization constraints with temporal properties, but it can be used for some SoD properties and constraints that can be expressed in RCL 2000 (cp. [Ahn99]). Analogously to model checking, complexity can cause problems (see subsection 3.3). Similarly to the USE tool and the graph transformation tool AGG, Alcoa might be employed as an authorization engine.

4.2.4 AGG

Representing the current state of an RBAC configuration as graph one can employ the graph transformation tool AGG to make sure that state transitions are consistency preserving w.r.t. a given RBAC policy or given authorization constraints (see subsection 3.4). Assuming the consistency of the initial state w.r.t. the given RBAC policy, it easily follows that all states of the RBAC configuration in question are consistent with the RBAC policy. Hence the AGG tool can be used as an authorization engine. Additionally, Koch et al. ([KMPP02]) present an algorithm to modify graph transformations such that they preserve the consistency w.r.t given graphical constraints. Of course, this algorithm is meant to be used for modifying graph transformation rules in order to make them consistency preserving w.r.t. a given RBAC policy. Analogously to Alcoa and the USE tool, there is no explicit representation of temporal structures and no support for deduction. Complexity is also an issue for AGG (cp. matching problem).

4.2.5 The USE tool

Since the USE tool is quite similar to Alcoa in terms of functionality, the conclusions in subsection 4.2.3 for Alcoa essentially apply also to the USE tool. The USE tool accepts UML as the input format for the models to be considered and OCL as the input language for the constraints to be evaluated. Both formalisms are rather popular in the industry. Thus the USE tool may be a well-suited tool for supporting policy designers especially in the initial design phase. Furthermore, the USE tool can be used as an authorization engine, a prototypic implementation has been realized in a master thesis.

4.3 Summary

First-order LTL, which has been encoded in Isabelle/HOL, is the most powerful and flexible of the considered formalisms. Since it provides inference rules, it is well-suited for proving security properties of RBAC policies. Propositional LTL can be used to describe regular RBAC configurations and is supported by NuSMV. The combination of first-order LTL and propositional LTL and/or Isabelle/HOL and NuSMV meets all essential requirements of validation in ORKA and is particularly appropriate for the investigation of security relevant workflow properties.
Graphs can be used to represent the states of an RBAC configuration. Consequently, state transitions can be expressed as graph transformations. Thus the graph transformation tool AGG can be used as an authorization engine, which has been demonstrated in a master thesis [Mig05]. According to [KMPP02], there is a deterministic terminating algorithm to make arbitrary graph transformations consistency preserving w.r.t. a given RBAC policy, which may increase the efficiency of authorization decision making. So graphs and graph transformations may be a good choice in order to construct an authorization engine.
The DTAC model is a quite general algebraic approach representing concepts as graph nodes offering a UML-like notation. Thus it may be a good starting point for policy design, although it does not offer inference rules or an explicit temporal structure. There is no known supporting

tool, but Alcoa, the USE tool or even AGG may possibly be employed for this purpose. The entity-relationship model is a formalism for defining arbitrary databases, which does not provide inference rules or a temporal structure. Regarding states of RBAC configurations as contents of a database, one may employ this formalism for the design of an authorization engine (as an alternative to graph-based formalisms). Adequate tool support would have to be found.
UML is a rather popular general-purpose modeling language, which can be used to specify the RBAC model [SA00]. OCL is part of the UML standard and allows the specification of authorization constraints. UML and OCL are supported by the USE tool. So UML/OCL supported by the USE tool may help policy designers in the initial design phase (as already mentioned the DTAC model might be used for the same purpose). Furthermore, the USE tool may serve as an authorization engine.
The described tools support only automated validation with the exception of the theorem prover Isabelle. As pointed out, Isabelle can accept a variety of formal calculi. Deduction is specifically facilitated by Isabelle. Furthermore, complexity is in general not a problem for Isabelle, which is rather dependent on human intervention.
The remaining tools do not support deduction, but are therefore essentially independent of human intervention. Among these tools Alcoa possibly achieves the highest degree of automation, i.e. needs the least instruction by the user. NuSMV is the only one of these tools which provides an explicit representation of temporal structures since it supports propositional LTL. All of these tools can only be applied to problems having a finite scope respectively a finite amount of possible solutions which can be checked by some automated algorithm. We close with the following diagrammatic overview (see table 1) of our findings.

Formalism	Form. semantics	Inf. Rules	Tool support	Temp. structure	Validation
First-order LTL	yes	yes	Isabelle	explicit	deduction-based
Prop. LTL	yes	no, but theoretically possible	NuSMV	explicit	automated and theoretically deduction-based
Graphs	yes	no	AGG	no but implicit extension possible	automated
DTAC model	yes	no	?	no but implicit extension possible	possibly automated
E-R model	yes	no	?	no	possibly automated
Alloy	yes	no, but theoretically possible	Alcoa	no	automated
UML, OCL TOCL	yes (under certain restrictions)	no	USE tool (for UML and OCL)	explicit (for TOCL)	automated

Table 1: Overview of formalism properties.

References

[Ahn99] G.-J. Ahn. *The RCL 2000 language for specifying role-based authorization constraints*. PhD thesis, George Mason University, Fairfax, Virginia, 1999.

[Ame04] American National Standard Institute (ANSI) for Information Technology. Role based access control. Technical Report ANSI INCITS 359-2004, ANSI, Feb. 2004.

[AS00] G.-J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226, November 2000.

[AS01] G.-J. Ahn and M.E. Shin. Role-Based Authorization Constraints Specification Using Object Constraint Language. In *Proc. of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, pages 157–162. IEEE, 2001.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: an open source tool for symbolic model checking. Technical report, January 01 2002.

[CGP99] E.M. Clarke, O. Grumberg, and A.D. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Che76] Peter Pin-Shan Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[DBTS04] 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*, pages 60–67, 2004.

[EU95] 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.

[FSG+01] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[Gol87] R. Goldblatt. *Logics of Time and Computation, Second Edition, Revised and Expanded*, volume 7 of *CSLI Lecture Notes*. CSLI, Stanford, 1992 (first edition 1987). Distributed by University of Chicago Press.

[HDF02] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting ocl. *Sci. Comput. Program.*, 44(1):51–69, 2002.

[JSS00] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 730–3, Limerick, Ireland, 4–11 June 2000. IEEE.

[K.L92] K.L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[KMPP02] M. Koch, L. V. Mancini, and F. Parisi-Presicce. A Graph Based Formalism for RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):332–365, August 2002.

[KW04] Anneke Kleppe and Jos Warmer. Would plato have built a unified modeling language? 2004. http://www.klasse.nl/papers/uml-plato-paper.pdf (2007-01-19).

[MDS03] 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.

[Mig05] L. Migge. Specification and Enforcement of Role-based Security Policies, 2005. Master Thesis, Universität Bremen.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[PN] L. Paulson and T. Nipkow. Isabelle. Look at URL http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[RG00] M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, pages 265–277. Springer, Berlin, LNCS 1939, 2000.

[RG02] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.

[Ric02] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[RLFK04] 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*, pages 115–124. ACM Press New York, USA, 2004.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

[SA00] Michael E. Shin and Gail-Joon Ahn. UML-Based Representation of Role-Based Access Control. In *WETICE '00: Proceedings of the 9th IEEE International Workshops on Enabling Technologies*, pages 195–200, Washington, DC, USA, 2000. IEEE Computer Society.

[SAGM05] 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.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[Sch03] A. Schaad. *A Framework for Organisational Control Principles*. PhD thesis, University of York,UK, 2003.

[SLS06] 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, June 2006. ACM Press.

[Sof] Software Technology Laboratory at Queen's University. UML 2 Semantics Project. http://www.cs.queensu.ca/ stl/internal/uml2/ (2007-01-19).

[SZ97] R. Simon and M. Zurko. Separation of duty in role-based environments. In *10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 183–194, June 1997.

[TER99] G. Taentzer, C. Ermel, and M. Rudolf. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter The AGG Approach: Language and Tool Environment. World Scientific, 1999.

[TJ00] Jonathon E. Tidswell and Trent Jaeger. Integrated constraints and inheritance in DTAC. In *RBAC '00: Proceedings of the Fifth ACM Workshop on Role-based Access Control*, pages 93–102, New York, NY, USA, 2000. ACM Press.

[TP01] Jonathon E. Tidswell and John M. Potter. A graphical definition of authorization schema in the DTAC model. In *SACMAT '01: Proceedings of the sixth ACM Symposium on Access Control Models and Technologies*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

[ZG02] P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 53–62, 2002.