Monitoring Database Access Constraints with an RBAC Metamodel: a Feasibility Study

Lars Hamann, Karsten Sohr, and Martin Gogolla

University of Bremen, Computer Science Department
D-28334 Bremen, Germany
{lhamann,sohr,gogolla}@informatik.uni-bremen.de

Abstract. Role-based access control (RBAC) is widely used in organizations for access management. While basic RBAC concepts are present in modern systems, such as operating systems or database management systems, more advanced concepts like history-based separation of duty are not. In this work, we present an approach that validates advanced organizational RBAC policies using a model-based approach against the technical realization applied within a database. This allows a security officer to examine the correct implementation – possibly across multiple applications – of more powerful policies on the database level. We achieve this by monitoring the current state of a database in a UML/OCL validation tool. We assess the applicability of the approach by a non-trivial feasibility study.

Keywords: Model checking for security; Models for security; Verification techniques for security properties; Security by design

1 Introduction

Modeling systems with languages like UML [14] and OCL [15] offers many advantages for the development process: Models allow developers to state, analyze and predict interesting characteristics of the system under study before an actual implementation is done, and models allow developers to specify the implementation of the intended system.

RBAC (Role-Based Access Control) is a well accepted approach for designing and implementing access management. Typically, many proposed approaches use RBAC in the system design phase in a forward engineering way. RBAC, however, also allows us to monitor access violations in a running system. Monitoring approaches can be employed for existing systems during running operations.

This contribution puts forward an RBAC modeling approach and is based on previous foundational work on an RBAC metamodel [11] and on runtime monitoring using UML and OCL [10]. This new approach combines both lines of work by integrating them and evaluating the applicability using relational database systems. We concentrate on databases rather than the application itself as access violations might also occur at this lower system level; the lower layer

might not respect the policies defined for the application level [2]. The evaluation is done by a feasibility study on a publicly available moderate sized database [20].

The rest of this paper is structured as follows. Section 2 introduces the essential RBAC concepts needed in our context and explains a running example. Section 3 sketches the UML and OCL tool USE (UML-based Specification Environment) and its monitoring features. Section 4 explains the details of monitoring RBAC on a relational database. Special emphasis is laid on dynamic separation of duty constraints. Section 5 discusses a feasibility study for our approach and shows that it works in a case study where some 100,000 database tuples are monitored and some 10,000 access operations on the tuples and violation of dynamic separation of duty rules are analyzed. Section 6 discusses related work. Section 7 summarizes the results and gives an outlook on future work.

2 Role-based Access Control

RBAC is widely used in organizations such as financial institutes or enterprises for access management. Users do not obtain permissions to access resources directly, but through roles. Roles often correspond to job functions that a user holds within her organization. The role concepts have been described in the RBAC ANSI standard [1]. RBAC comprises the sets *Users*, *Permissions*, and *Roles*, as well as the relations *UA*, *PA*, and *RH*. *UA* is a many-to-many relation which represents the roles assigned to users ("user assignment"). The assignment of permissions to roles is expressed by the many-to-many relation *PA*. Furthermore, permissions are often seen as pairs of resources and actions, e. g., the action "approve" is allowed to be performed on the resource "cheque".[1] *RH* describes a role hierarchy relation on the set of *Roles*, i.e., roles can inherit permissions from other roles.

The aforementioned RBAC sets and relations are shown in Fig. 1 where we present a UML-based metamodel of RBAC. For example, the *Users* and *Roles* sets are represented by the `User` and `Role` classes; the *UA* relation is expressed by the association between both classes. Please note that permissions are represented by an association class between the `resource` and `action` classes.

RBAC also supports advanced access control concepts, such as role hierarchies and role-based authorization constraints. The role hierarchies are represented by a senior/junior association from the `Role` class to itself in Fig. 1. A typical example of a role hierarchy relation is given by the roles `Cashier` and `Cashier Supervisor` where the former role is junior to the latter.

Authorization constraints allow a security officer to express organizational rules. The most well-known kind of authorization constraint is separation of duty (SoD). SoD prevents a user from committing fraud by splitting tasks into several parts, which must be executed by different users [18]. Two forms of SoD are usually distinguished, namely, static and dynamic SoD. Static SoD

[1] In the literature on access control, usually the terms "object" and "operation" are used instead of "resource" and "action". However, we felt that the former terms could be mixed with the notions of object and operation in UML.

is often expressed in terms of mutually exclusive roles, e.g., the roles `Clerk` and `Supervisor`. Static SoD is enforced at administration time, i. e., when a user is assigned to a role. In contrast, dynamic SoD is enforced at runtime. For example, a banking clerk who has approved a cheque is not allowed to validate it. Such conditions often need the access history for access decisions [18]. As a distinguishing feature, the RBAC metamodel given in Fig. 1 also supports this kind of dynamic SoD, which is represented by the class `DynamicSoD`.

Figure 2 visualizes our concept of modeling dynamic SoD and is meant for didactic purposes. After a user has applied the action `preAct` to the resource property `preProp` belonging to a resource of type `preT`, she is not permitted to apply `postAct` to the resource property `postProp` of a resource of type `postT` (dynamic SoD constraint). In the aforementioned example, the resource types `preT` and `postT` equal "cheque". The preaction is "approve", whereas the postaction is "validate". The properties are then "approved" and "validated", respectively.

3 USE

3.1 Validation and Verification with USE

Modeling features and their analysis through validation and verification is supported by the tool USE (UML-based Specification Environment) [7]. Within USE, UML class, object, statechart, sequence, and communication diagrams extended with OCL are available. USE assists the developer in order to validate and verify model characteristics. Validation and verification can be realized in USE by employing a model validator based on relational logic and SMT solvers. Model properties to be inspected include consistency, redundancy freeness, checking consequences from stated constraints, and reachability. These properties are handled on the conceptual modeling level, not on an implementation level. Employing these instruments, central and crucial model characteristics can be successfully and efficiently analyzed and checked.

3.2 Monitoring with USE

The USE Monitor project was started as a USE plug-in to support runtime verification of Java applications [10]. The monitor allows a developer to attach to a running application, take a snapshot of its current state and validate this snapshot against defined constraints. Using the monitor, an application can be verified at a more abstract level than the code, because the used model can be a small part of the overall system, by dropping unnecessary details. For example, a huge inheritance hierarchy can be compressed to those super classes required for the validation task. After an initial snapshot has been taken, the suspended application can be resumed to monitor changes in the system. While listening to change events, like the creation of new instances or operation calls, constraints defined in the model are validated, e. g., when the monitor receives an operation call event to an operation that is considered in the model, it evaluates the

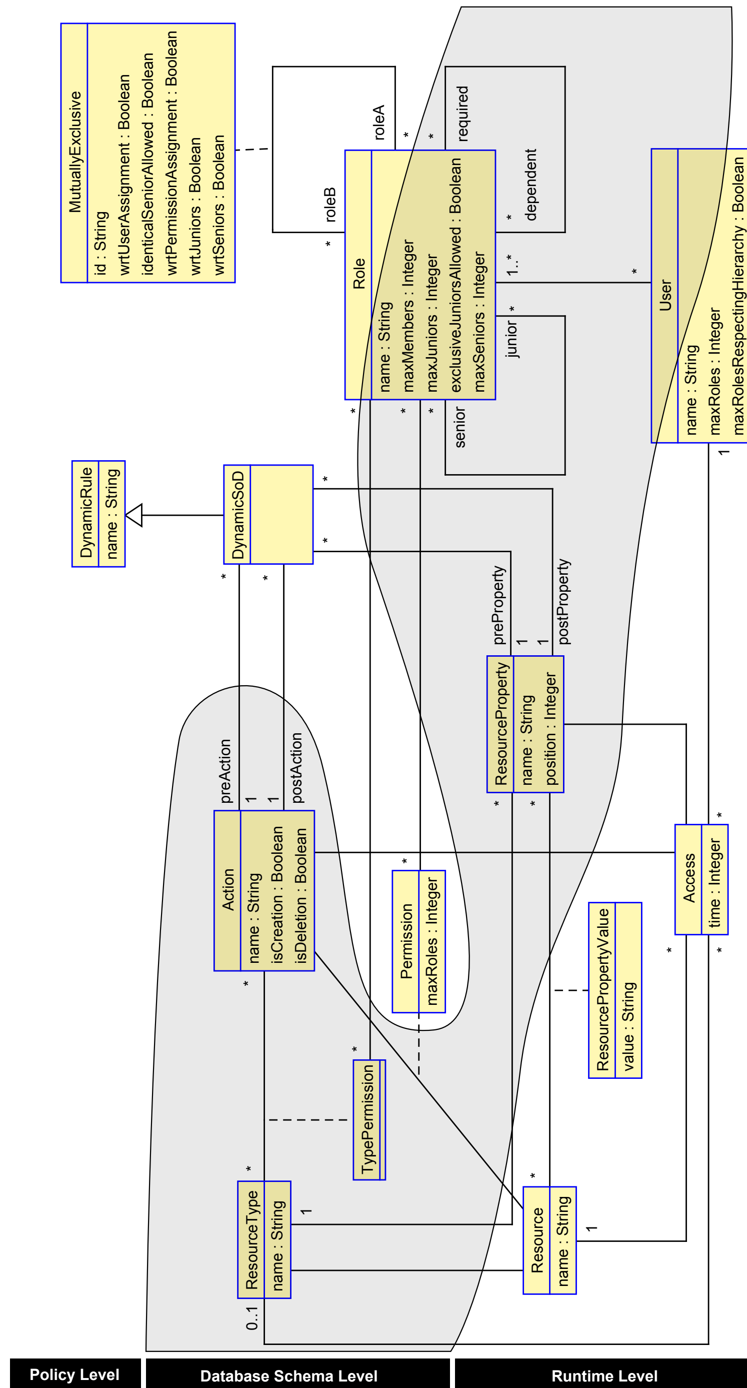


Fig. 1. Excerpt of the RBAC metamodel

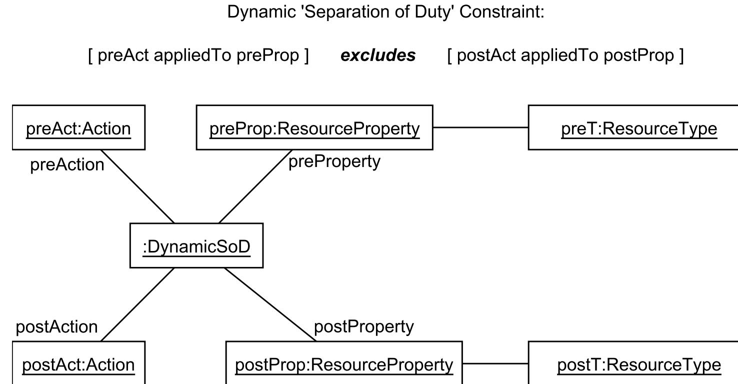


Fig. 2. Realizing dynamic SoD in our RBAC metamodel

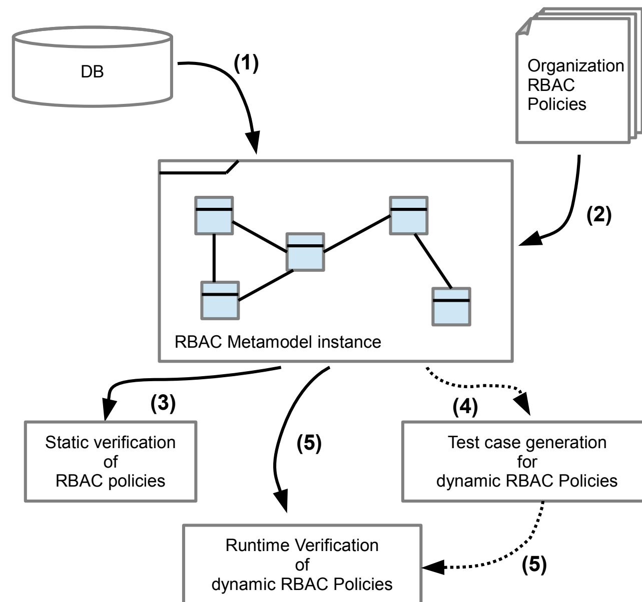
defined preconditions and, if present, possible state machine transitions for this operation. The user of the monitor is informed immediately about any unexpected behavior of the application, i. e., about model constraint violation. While the first version of the monitor was limited to the Java runtime environment, the current version allows for an easy integration of other target platforms. This is achieved by using so-called adapters. These adapters detach the monitor from a concrete target platform by hiding the concrete communication between the target platform and the monitor. Moreover, an adapter can map non object-oriented platforms (i. e., relational databases) to the object-oriented world of UML and OCL as it is done in this work.

4 Monitoring RBAC Constraints with UML and OCL Models

This section describes our approach of verifying RBAC policies defined on an abstract level against low level implementations in a step-wise manner. Figure 3 shows an overall picture of it. A central part of this approach is the RBAC metamodel described in detail in [11].

An excerpt of a slightly modified and extended version of this metamodel can be seen in Fig. 1. A central extension was done by integrating type information into the metamodel by adding the classes `ResourceType` and `ResourceProperty`. The RBAC specification is very general about the concrete meaning of a resource, but for our approach we need to distinguish between the permissions on a type, in our example a table, and the permissions on access to an instance, i. e. a table entry, of these types.

Before we discuss the steps in detail, we briefly describe the overall workflow. In Step 1, the USE monitor is utilized to retrieve any information that is relevant to the overall RBAC policies from a database. Relevant information include



Step 1: Retrieve RBAC relevant information from database
Step 2: Apply organization RBAC policies
Step 3: Validate DB permissions against RBAC policies
Step 4: Generate test cases for dynamic aspects
Step 5: Runtime verification of dynamic RBAC policies

Fig. 3. Overview of the verification process

defined roles, tables, columns and database users. These elements are created within the USE tool as objects according to the RBAC metamodel.

After this, the developer applies the organization RBAC policies to the information extracted from the database in Step 2. These organization policies contain rules that cannot be specified in the database. Examples are static and dynamic SoD constraints, which cannot be expressed in most database systems. Simple mismatches between the organization policies and the database security configuration, like missing roles or resources can already be identified in this phase of the process.

In Step 3, basic and extended verification jobs of the organization policies can be run. The basic verification evaluates the defined invariants on the metamodel in order to discover violations of static RBAC constraints, e. g., the violation of mutually exclusive roles defined in the organization policies. Extended verification applies more powerful model finding techniques to verify assumptions about the configuration, e. g., whether a given workflow is executable with the current security settings.

The same model finding approach is used in the optional step 4 to generate test cases for validating dynamic RBAC policies. It can be used to generate scenarios (workflows) that violate a given dynamic policy. These scenarios can be used to test if the applications using the database take into account the organizational dynamic RBAC policies.

For Step 5 the USE monitor is utilized to listen to database changes. After interesting change events, USE can validate the organizational dynamic RBAC policies at runtime and report violations to the developer. This step allows an application-independent runtime verification of dynamic policies. The test cases generated in Step 4 can be used as a test driver or the monitoring is done during "ordinary work", if the performance impact of the monitoring is not an issue.

After this summary of the overall process, we now discuss each step in detail.

4.1 Retrieve RBAC Relevant Information from a Database

To be able to verify more general security settings, i. e., RBAC policies, against the concrete permissions present in the database, the concrete permissions need to be available to a verification tool. For our approach we developed an SQL RBAC adapter that reads the database schema, the defined permissions and – if required – the current database state. Table 1 shows the currently used mappings from relational database elements to the metaclasses of our RBAC metamodel.

Table 1. Element mapping from Database elements to RBAC metamodel classes

Element	Catalog Item	→ RBAC metamodel class
Table	information_schema.tables	→ ResourceType
Column	information_schema.columns	→ ResourceProperty
User*	pg_user	→ User
Role*	pg_group	→ Role
Permission Kind	Fixed values (INSERT, ...)	→ Action
Table Permission	information_schema.table_privileges	→ TypePermission
Tuple*		→ Resource
Value*		→ ResourcePropertyValue

*These elements require special treatment and are described in Sect. 4.1.

This table is just a brief overview to illustrate the idea of the mapping. It does not contain all relevant information. For example, we do not provide information about the role membership and role hierarchy. For this kind of relation, the mapping is typically done by using foreign key information of the corresponding relations. The information for the metaclasses `ResourceType`, `ResourceProperty` and `TypePermission` can be retrieved by querying the database information schema as it is defined in the SQL standard (c. f. [8]). Each returned tuple is considered as a new instance of the mapped metaclass. Figure 5 shows the object diagram that is extracted by the monitor after it was applied to a database containing the table shown in Fig. 4.

cheque	nr [PK] text	amount numeric(10,2)	approved integer	validated integer
1	100005	150.00	-1	0
2	100006	120.00	-1	-1

Fig. 4. Table *cheque* with sample data

In the upper part of this figure, the table structure can be seen. Directly below the structure of the table, its content is shown. In the example the content of the table consists of two entries. The cheque with the number 100005 and `amount` 150.00 has been approved (the property `approved` $\neq 0$) but not validated (the property `validated` $= 0$). The other cheque with number 100006 about 120.00 has already been validated and approved. The values of a table can be extracted by taking into account the information about the table schema and constructing corresponding SQL statements. Since not all data contained in a database are required for the following tasks, adequate filters could be used to reduce the overall size of the resulting object diagram. Specifically, the applicability of model finding during some tasks depends on this reduction.

The permissions on resource types, i. e., tables, can be retrieved by querying the default information scheme, too. However, the concrete users and roles are not easy to query using the SQL standard. In our work, we used the views `pg_roles` and `pg_users` which are specific to the database system PostgreSQL[2]. Since the possible actions on tables are defined in the SQL standard, they can be defined beforehand. In our example, the defined permissions (instances of association class `ResourcePermission`) and their assignments to roles (represented as links) can be seen on the left side of Fig. 5. One can retrieve that the role `Supervisor` has the following permissions on table `cheque`: `DELETE` (object `tp3`), `UPDATE` (`tp2`), and `SELECT` (`tp4`), whereas, the role `Clerk` has the permissions `SELECT` (`tp4`), `INSERT` (`tp1`), and `UPDATE` (`tp2`).

4.2 Apply Organizational RBAC Policies

In this step, the developer enriches the monitored information with RBAC policies from the organization that cannot be expressed in the used database system. For example, the RBAC concept of dynamic SoD with respect to access history is unsupported in database systems.

To apply these more general policies, the developer needs to modify the instance of our RBAC metamodel read in Step 1. She can set attribute values of already present metamodel elements, such as roles, define new rules by creating rule objects and, link instances to fit the organization needs. Please note that a developer only needs to configure instances of provided rules. She does not need to write policy rules in OCL, since these rules are already defined in the metamodel. Further, we only show one example of a policy (DynamicSoD), but

[2] Using Microsoft SQL Server the corresponding system view is `syslogins`. The column `issqlrole` determines if an entry is a role or a user.

Fig. 5. The table shown in Fig. 4 as an instance of the RBAC metamodel, generated with the USE tool

other policies can be integrated easily by a metamodel designer into the RBAC metamodel, if required. After these policies have once been defined, they can automatically be applied in another verification session by our tool chain.

Suppose that the two roles `Clerk` and `Supervisor` are defined as mutually exclusive (static SoD) by organizational rules. For this, a developer needs to create an instance of the association class `MutualExclusive` between both roles.

More advanced policies can be defined by creating instances of the class `DynamicSoD`. This class enforces a dynamic SoD constraint. An example of such a dSoD constraint is shown in Fig. 6. This figure is a more specific version of the one shown in Fig. 2. Using natural language, this dSoD constraint states: *"If a user approves a cheque, she is not allowed to validate it."* For the database schema model, this rule is defined by specifying the action `UPDATE` and the resource property `approved` as the preceding access and the same action together with the resource property `validated` as a forbidden posterior access.

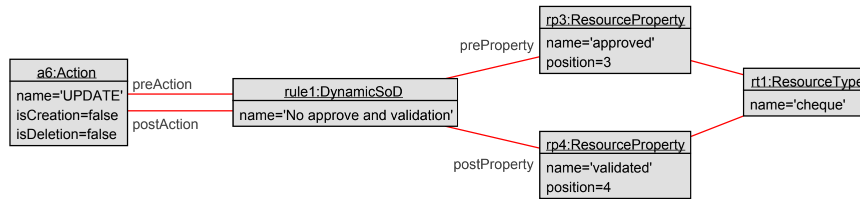


Fig. 6. Modeled dynamic separation of duty policy

In the following sections we describe how the policies explained in this section are used to validate the overall security settings. Before this, we would like to mention that already at this stage in the process, mismatches between the database configuration and the policy level can be detected. A possible mismatch is the absence of a role in the database, which is used in the organization policies. The same holds for resource types and properties. These dependencies between elements at the policy level and elements at the database schema level can be seen in Fig. 1. Classes and associations that cross the highlighted levels are used in both levels.

4.3 Validate Database Permissions against RBAC Policies

After the concrete database settings have been merged successfully with the RBAC policies, a developer can now verify the validity of the policies w. r. t. the current database configuration. Note that in this step no dynamic policies can be verified because the monitored data contain no information about past events. However, some useful static policies can be verified. As most database systems do not allow an administrator to specify a maximum number of users for a role, e. g., only two supervisors are allowed in a given company, these organization policies can now be verified. Our RBAC metamodel contains invariants specified

in OCL for several extended RBAC policy types, including the maximum number of roles a user can be a member of, mutually exclusive roles and role hierarchies.

Using our validation and verification tool USE, violations of these invariants can be discovered. USE evaluates the given invariants against the system state which was created in the previous step. If a constraint fails, USE provides a rich set of functionality to discover the violating elements. The outcome of this phase can be used to give advice to database administrators how to change the security settings within the examined database.

Another question which can be answered in this step is if a given workflow is executable w. r. t. current database permissions. For our example, we want to know, if a cheque can be approved and afterwards be validated. For this small example a database administrator might directly see that the workflow is executable. However, for more complex workflows automatic verification techniques are needed. Our toolchain supports this task by using the model validator plug-in described in Sect. 3.2. To do so, a developer needs to specify the workflow to validate by means of declarative assumptions that need to be fulfilled. The workflow to validate a cheque requires to describe a correct execution, i. e., it has to be specified that a cheque is approved and validated. This can be done by defining an invariant that enforces two access actions to the same resource of type named *cheque*: one for the update of the resource property `approved` and one for the update of the property `validated`. This invariant together with other settings, like bounds for the number of instances for each type, are provided to the model validator, which then tries to find a valid object diagram w. r. t. the default RBAC constraints and the additional invariant. The model validator uses the previously created object diagram as a starting point, a so-called partial solution. If the model validator finds a solution within the provided bounds, the workflow is executable under the current permissions of the database. If it does not find a solution, one cannot directly state that the workflow is not executable, because the search space of the model validator is limited by the provided bounds. This means, a valid solution might exist outside the configured bounds. A developer can now increase the bounds until it is likely that no valid solution exists. Hints to the developer why a given setup is not satisfiable can also be provided by the model validator.

4.4 Generate Test Cases for Dynamic Aspects

The model validator can be used for test case generation, too. For this task, the approach described in the previous section is slightly changed: instead of providing information about a valid workflow, the model validator is now used to find a solution that does not fulfill a given constraint, e. g., the invariant or invariants that define a dynamic SoD policy. This negation of invariants is directly supported by the model validator, since it is useful for several examination tasks in the context of model finding.

Figure 7 shows a resulting test case using the cheque example. The model validator extended the object diagram shown in Fig. 5 by several instances. Note that we do not show all instances again. Only the relevant ones for the invalid

workflow are displayed. All other are hidden in the object diagram, but are present in the used system state. In detail, the model validator added three access objects each representing an access to a resource or resource property of the resource type `cheque` by the user `bob`. The first access (`time`=2) creates resource `r1` using the INSERT action (`a3`). Afterwards, the access sequence violating the dynamic SoD rule shown in Fig. 6, which updates the properties `approved` (at time 28) and `validated` (at time 32) occur. These generated test cases can be seen as basic execution sequences that violate a given high-level policy. The concrete workflow execution, possibly using different applications, depends on the application landscape present. Therefore, a security officer needs to map the generated basic access operations to a business workflow to execute the test case.

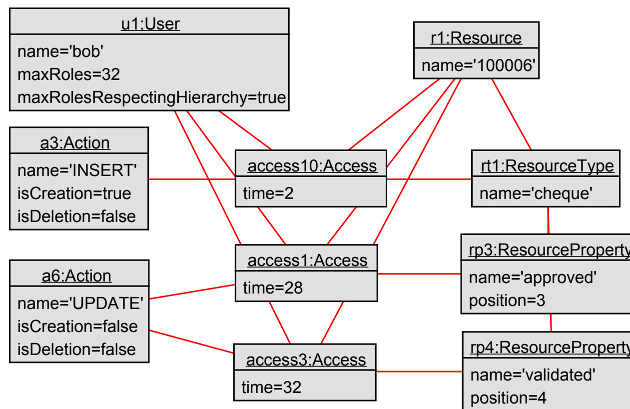


Fig. 7. Generated test case for an invalid dSoD policy

4.5 Runtime Verification of Dynamic RBAC Policies

Until now, we only considered the database state as it is. However, using the monitor plug-in dynamic verification could be applied, too. For this, the database management system needs to provide a notification mechanism that allows an application to register for interesting events. These events include statements for the actions `INSERT`, `UPDATE`, and `DELETE`. An example of such mechanism are the Microsoft Notification Services [12] provided by SQL Server 2005. Starting with SQL Server 2008 these services are integrated into the SQL Server Reporting Services [13]. Using PostgreSQL a notification infrastructure can, for example, be built by using the non-standard SQL command `NOTIFY` [19]. Both mechanisms have in common that the notifications are sent asynchronously. Since our approach is designed to identify policy violations and not to prevent them, this asynchronous behaviour fits well. Only the sequence of access must be correct. In

addition, using asynchronous notifications reduces the overhead for the running applications, since the validation can be done independently.

Using such an appropriately configured infrastructure, the monitor could react on events by notifying the USE tool about changes. USE could then validate all defined RBAC policy rules based on its knowledge about the history of executed commands. This history would be built incrementally starting at the first received command. Commands executed before cannot be considered. Therefore, to verify a complete workflow, this workflow needs to be executed as a whole during the runtime verification task. One benefit of this runtime verification approach would be that the correct implementation of RBAC policies within a single application or across multiple applications using the same data source can be checked, because the evaluation of the policies is done on the database layer.

5 Feasibility Study

Until now, we only showed small examples adequate for presenting the overall ideas and explained concepts of a possible runtime verification task. To get an early impression about the database sizes that our approach can handle, we extracted database information of various sizes and generated 10,000 access operations. Afterward, we ran a complete validation of the structure, i. e., we checked the extracted object diagram against the multiplicity constraints and the (OCL) invariants present in our metamodel. Further, we ran these validations using different numbers of RBAC policy rules. Starting with seven rules and ending with 28. Table 2 summarizes our results. All performance evaluations, except the second row, were executed on a workstation powered by a Pentium DualCore E5300 with 2.6GHz with 4GB RAM running Ubuntu 14.04LTS. USE was executed using Oracle JRE 1.7 with an allowed heap memory size of 2GB. The results in the second row of the table are collected from a notebook with a similar configuration, but running Windows 7 32bit, which only allows a maximum Java heap size of some 1GB. The complete RBAC metamodel including detailed information about the evaluation can be found online [9].

Table 2. Performance evaluation

				Validation time with n policy rules							
				Structure				Invariants			
#Rec.	#Inst.	#Links	read	7	14	21	28	7	14	21	28
74,964	638,897	75,009	52s	1s	1s	1s	1s	4.7s	8.8s	12.1s	15.5s
(Windows 7)			(53s)	(1.6s)	(1.6s)	(1.6s)	(1.6s)	(6.4s)	(11.5s)	(17.4s)	(23.1s)
89,222	769,329	89,267	60s	1.4s	1.4s	1.3s	1,2s	5s	9.6s	12.7s	16.8s
112,366	986,206	112,409	78s	1.6s	1.6s	1.5s	1.5s	4.7s	8.7s	12.8s	17.1s

In summary, we can state that our approach is able to validate RBAC policies in about 20 seconds on a moderate sized database (with about 100,000 tuples taking into account 10,000 access operations) and including on the RBAC side about 30 dynamic SoD rules (following the template in Fig. 2 and instantiated in example form in Fig. 6). Given the presented results, one can see that both the creation of the database state (column *read*) and the validation of the invariants grow linear. Further, the measured durations are still acceptable when taking into account that they represent a validation of the whole system state.

Currently, the memory consumption of USE in combination with the RBAC monitor is quite large. Using 1GB heap space, only the first database state containing roughly 75,000 records is manageable. This number can be increased, since the RBAC monitor currently keeps a copy of all read database rows for performance reasons. USE as a standalone application or used as a library can still handle more instances.

6 Related Work

Our approach builds on an RBAC metamodel [11] that we have extended with the notions of `ResourceType` and `ResourceProperty` (see Section 4). Several other works exist that model RBAC policies with UML and OCL [4, 3, 17, 6]. Some of those approaches also target at validating RBAC policies, e. g., [3]. However, none of those works deal with runtime monitoring and they do not support dynamic SoD constraints based on the access history.

Only simple SoD concepts have been implemented in database management systems (DBMS) until now, such as mutually exclusive roles w.r.t. role membership (static SoD) or role activation (simple dynamic SoD). Advanced role-based authorization constraints, such as history-based dynamic SoD, are not supported. RBAC concepts for relational databases have also not been discussed much in scientific literature in recent years. In older work, Ramaswamy and Sandhu discuss RBAC concepts that are supported by commercial DBMS [16]. Later, Bertino and Sandhu came to the conclusion that commercial DBMS use only limited RBAC concepts [5]. Our experience with current versions of DBMS still supports this statement as mentioned before.

7 Conclusion

We have employed a UML and OCL model monitor for RBAC in relational databases. The approach can detect, for example, the violation of dynamic mutually exclusive roles specified as organizational policies. We have performed a feasibility study where a moderate relational database with some 100,000 tuples together with about 10,000 database access operations have been monitored in our tool USE. The approach allows one to check the consistency of the specified policies with actual workflows. We have concentrated on dynamic SoD constraints with respect to an access history.

The current approach is only a first step towards monitoring RBAC policies during runtime. Future work should study further types of RBAC constraints which can be captured by our metamodel. Currently, the formulation of the organizational policies in form of object diagrams is not user-friendly. A better syntactical presentation should be developed. The utilization of the model validator and its features could also be improved, and further steps in the monitoring process, like schema and constraint extraction, could be automatized.

References

1. American National Standards Institute Inc.: Role Based Access Control (2004), ANSI-INCITS 359-2004
2. Anderson, R.J.: Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley Publishing, 2 edn. (2008)
3. Basin, D.A., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Information & Software Technology 51(5), 815–831 (2009)
4. Basin, D.A., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Trans. Softw. Eng. Methodology 15(1), 39–91 (2006)
5. Bertino, E., Sandhu, R.: Database Security-Concepts, Approaches, and Challenges. IEEE Trans. Dependable Secur. Comput. 2(1), 2–19 (Jan 2005)
6. Fernández-Medina, E., Piattini, M.: Extending OCL for secure database development. In: Proc. of UML 2004. LNCS, vol. 3273, pp. 380–394 (2004)
7. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Sci. of Comp. Prog. 69, 27–34 (2007)
8. Gulutzan, P., Pelzer, T.: SQL-99 complete, Really – An Example-Based Reference Manual of the New Standard. R&D Books (1999)
9. Hamann, L., Gogolla, M., Sohr, K.: RBAC meta-model and detailed evaluation results, `http://www.db.informatik.uni-bremen.de/publications/intern/RBACEvaluation.use`, last visited: 03/20/2014
10. Hamann, L., Hofrichter, O., Gogolla, M.: OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In: Proc. 8th European Conf. on Modelling Foundations and Applications. pp. 384–399. Springer, LNCS 7349 (2012)
11. Kuhlmann, M., Sohr, K., Gogolla, M.: Comprehensive Two-Level Analysis of Static and Dynamic RBAC Constraints with UML and OCL . In: Proc. Secure Software Integration and Reliability Improvement (SSIRI'2011). pp. 108–117. IEEE (2011)
12. Microsoft: SQL Server Notification Services, `http://technet.microsoft.com/en-us/library/ms172483%28v=sql.90%29.aspx`, last visited: 02/05/2014
13. Microsoft: SQL Server Reporting Services (SSRS), `http://technet.microsoft.com/en-us/library/ms159106.aspx`, last visited: 02/05/2014
14. UML Superstructure 2.4.1. Object Management Group (OMG) (Aug 2011), `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`
15. Object Constraint Language 2.3.1. Object Management Group (OMG) (Jan 2012), `http://www.omg.org/spec/OCL/2.3.1/`
16. Ramaswamy, C., Sandhu, R.: Role-Based Access Control Features in Commercial Database Management Systems. In: Proc. of 21st National Information Systems Security Conference. pp. 503–511 (1998)

17. Ray, I., Li, N., France, R.B., Kim, D.K.: Using UML to visualize role-based access control constraints. In: Proc. of the 9th ACM Symp. on Access Control Models and Technologies. pp. 115–124 (2004)
18. Simon, R., Zurko, M.: Separation of duty in role-based environments. In: 10th IEEE Computer Security Foundations Workshop (CSFW '97). pp. 183–194 (1997)
19. The PostgreSQL Global Development Group: PostgreSQL 9.3.2 Documentation: NOTIFY, `http://www.postgresql.org/docs/9.3/static/sql-notify.html`, last visited: 02/05/2014
20. Treat, R., Mohan, V.: pgFoundry: Sample Databases, dellstore2, `http://pgfoundry.org/projects/dbsamples/`, last visited: 03/20/2014