Software Verification and Validation Laboratory: A Comprehensive Modeling Framework for Role-based Access Control Policies

Ameni Ben Fadhel, Domenico Bianculli and Lionel Briand

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

TR-SnT-2014-15

ISBN: 978-2-87971-137-9

April 22, 2015

Version 1.1

A Comprehensive Modeling Framework for Role-based Access Control Policies

Ameni Ben Fadhel Domenico Bianculli Lionel Briand

May 13, 2015

Abstract

Prohibiting unauthorized access to critical resources and data has become a major requirement for enterprises. Access control (AC) mechanisms manage requests from users to access system resources; the access is granted or denied based on authorization policies defined within the enterprise. One of the most used AC paradigms is role-based access control (RBAC). In RBAC, access rights are determined based on the user's role, e.g., her job or function in the enterprise.

Many different types of RBAC authorization policies have been proposed in the literature, each one accompanied by the corresponding extension of the original RBAC model. However, there is no unified framework that can be used to define all these types of RBAC policies in a coherent way, using a common model. Moreover, these types of policies and their corresponding models are scattered across multiple sources and sometimes the concepts are expressed ambiguously. This situation makes it difficult for researchers to understand the state of the art in a coherent manner; furthermore, practitioners may experience severe difficulties when selecting the relevant types of policies to be implemented in their systems based on the available information. There is clearly a need for organizing the various types of RBAC policies systematically, based on a unified framework, and to formalize them to enable their operationalization.

In this paper we propose a model-driven engineering (MDE) approach, based on UML and the Object Constraint Language (OCL), to enable the precise specification and verification of such policies. More specifically, we first present a taxonomy of the various types of RBAC authorization policies proposed in the literature. We also propose the GemRBAC model, a generalized model for RBAC that includes all the entities required to define the classified policies. This model is a conceptual model that can also serve as data model to operationalize data collection and verification. Lastly, we formalize the classified RBAC policies as OCL constraints on the GemRBAC model. To facilitate such operationalization, we make publicly available online the Ecore version of the GemRBAC model and the OCL constraints corresponding to the classified RBAC policies.

1 Introduction

Prohibiting unauthorized access to critical resources and data has become a major requirement for enterprises. Access control (AC) mechanisms manage requests from users to access system resources; the access is granted or denied based on the authorization policies defined within the enterprise. Access control systems can be grouped into three categories [37]: *discretionary* (DAC), *mandatory* (MAC), and *role-based* (RBAC). In DAC, access rights are directly assigned to each user; moreover, a user is the only entity that can control the access to her own object(s), by assigning access rights to other users. In the second category, MAC, the access rights are determined according to mandated regulations stated by a central authority. In RBAC, access rights are determined based on the user's role, e.g., her job or function, as well as on the permissions assigned to each role. By decoupling users from permissions, RBAC simplifies the administration and the deployment of access control policies in large enterprises. In the rest of this paper, we focus on RBAC, since it has become the de facto standard for access control in enterprise systems [29].

The concept of role-based access control was initially proposed by Sandhu et al. in 1996 [38]; later on, the various initial proposals of RBAC models were consolidated into a unified standard model for RBAC, proposed by the NIST [36]. The basic RBAC model is composed of: 1) entities, corresponding to users, roles, sessions, and permissions; 2) relations among these entities. A user is allowed to execute a set of permissions that corresponds to the role(s) assigned to her; in other words, a role maps each user to a set of permissions. A session maps each user to the set of her active role(s).

RBAC supports three security *principles*: least privilege, data abstraction, and separation of duty. The least privilege principle requires a user to be authorized to execute only the minimal set of permissions needed for a given task, as determined by her role. The data abstraction principle is satisfied by abstracting low-level operations (e.g., the *read* and *write* operations provided by the operating system) with high-level operations defined for each business object in the system (e.g., updating the list of employees). The separation of duty principle states that no user should be given sufficient permissions to misuse the system. Although these principles are supported by RBAC, they are not automatically enforced by a system implementing RBAC: additional authorization constraints, called also *policies*[1], have to be defined to restrict the user's access.

Various types of authorization constraints have been proposed in the literature. For instance, cardinality constraints represent a bound on the number of roles and sessions to which a user can be assigned. Prerequisite constraints are a precondition on user-role assignment, stating that a user can be assigned to a role only if the user is already a member of another role. Separation of duty constraints (SoD) define a mutual exclusion relation among roles, permissions, or users. Dually, binding of duty (BoD) constraints define a correlation among a set of operations that must be performed by the same user. Delegation constraints allow a user to temporarily transfer a set of permissions associated to her role to another user. Context constraints restrict a user from performing an action depending on her current location or on the time at which the action should happen.

Various extensions of the original RBAC96 model have been proposed to support these different types of constraints. However, there is no unified framework that can be used to define all these types of authorization constraints in a coherent way, using a common model. The lack of a unified framework makes difficult for practitioners to understand, select among, and implement the different types of policies proposed in the literature.

In this paper we survey and classify the various types of RBAC authorization constraints proposed in the literature, and describe the different facets that characterize each type of constraint. We also review the different extensions of the original RBAC model that have been proposed in the literature to support the various types

[1]In the rest of this paper, we will use the terms "(authorization) constraints" and "policies" interchangeably.

of constraints. The main result of this review is that none of the proposed models can support all the constraints included in our classification. To address this limitation, we propose the GemRBAC model, a *Generalized Model for RBAC* that includes all the conceptual entities required to define the classified constraints. We then specify in an unambiguous manner all types of constraints to enable their operationalization. The specification follows a model-driven approach, based on UML and the Object Constraint Language (OCL): the classified constraints are formalized as constraints expressed with OCL on the GemRBAC model. This formalization brings three benefits: 1) it enables practitioners to select and make use of the various policies in a precise manner, based on the GemRBAC model; 2) it lays the ground for the practical verification of such policies, both at design time and at run time, based on UML modeling tools and OCL checkers (such as Eclipse OCL [15]); 3) it shows the expressiveness of the GemRBAC model, since it can accommodate all types of constraints included in our classification.

More specifically, the main contributions of this paper are: 1) a taxonomy, classifying the main RBAC policies proposed in the literature; 2) the GemRBAC model, which is a generalized model for RBAC that includes all the entities required to define the policies classified in the taxonomy; 3) the formalization, as OCL constraints on the GemRBAC model, of the RBAC policies included in the taxonomy; these constraints have been made publicly available, together with an Ecore [14] version of the GemRBAC model, at `https://github.com/AmeniBF/GemRBAC-model`.

The rest of the paper is organized as follows. Section 2 discusses the motivations of this work. Section 3 describes the original RBAC conceptual model. Section 4 presents a taxonomy of the various types of RBAC constraints proposed in the literature. Section 5 illustrates the various extensions to the original RBAC model. Section 6 introduces the GemRBAC model. Section 7 presents the specification of RBAC policies using OCL constraints defined on the GemRBAC model. Section 8 discusses, with an example, the application of the proposed model for the verification of RBAC policies. Section 9 discusses the related work while section 10 concludes the paper and provides directions for future work.

2 Motivations

RBAC is an access control mechanism that defines rules for authorizations and access restrictions for each role within an organization. Such a policy is required to specify access rights according to an individual's job or function (i.e., her role); unlike traditional access control, rights are not assigned to a user according to her identity. RBAC is available in some, security-oriented variants of Unix-like operating systems, as well as in modern database management systems. These systems implement a subset of the NIST RBAC model [36], based on the initial proposal of Sandhu et al. [37].

As we will see in the next sections, the original RBAC model supports a limited number of different types of authorization constraints, which cannot fulfill the expressiveness requirements that have emerged in the recent years in modern organizations. Examples of these new requirements are supporting delegation and revocation of permissions, and enabling access control policies based on the spatio-temporal context of users.

To fill this gap, researchers have proposed several extensions of the original RBAC model, to support the definition of new types of constraints (see sections 4 and 5). Though this work opens new possibilities for applying RBAC in modern enterprise systems, it is not easy to exploit in its current form. Indeed, these types of constraints and their corresponding models are scattered across multiple sources, are defined using different formalisms, and sometimes the concepts are expressed in an ambiguous manner.

This situation is very impractical for practitioners who want to select the relevant types of policies to be implemented in their systems. Moreover, they are faced with several models, often partially overlapping with each other, but with slightly different semantic variations. Furthermore, to the best of our knowledge, there is no model that can express *all* the type of constraints that we have identified in our survey. Last, scattered and heterogenous models make it difficult for researchers to understand the state of the art in a coherent manner.

We contend there is clearly a need for organizing the various types of RBAC authorization constraints systematically, based on a unified framework. The goal would be to formalize these constraints in such a way as to enable and facilitate their operationalization. This is the reason for which we propose the GemRBAC model, as a unified RBAC model that captures *all* the types of constraints found in the literature. Furthermore, for each type of authorization constraint, we define its formalization using OCL. By using such a common, standardized, and well-supported language, we not only facilitate the precise understanding of such constraints but we also facilitate their operationalization through industry-strength tools.

3 The original RBAC conceptual model

The original RBAC conceptual model, proposed in 1996 by Sandhu et al. [38], is composed of *users*, *roles*, *sessions*, and *permissions*; figure 1 illustrates the different components of this model and the relations between them. According to Sandhu et al., a role can be seen, at the same time, both as a collection of permissions and as a collection of users. A role can be assigned to one or more users via a *user-role assignment* relation. A *role-permission assignment* relation maps each role to one or more permissions. A session is a mapping of one user to a subset of the roles that have been assigned to her; this mapping *activates* the role(s) for a certain user. A permission allows a user to perform some operation(s) on some resource(s) of the system.

A role can be inherited using a *role hierarchy* relation, as shown in figure 2. A role can have one or more juniors (sub-roles) denoted by an arrow. For instance, r_2, r_3 and r_4 are juniors of r_1. In addition to its assigned permissions, r_2 inherits all permissions from its ancestor r_1. Moreover, a junior role can have one or more ancestors (senior roles). As shown in figure 2, r_5 inherits not only the permissions of r_2 but also the ones of r_3.

A set of authorization constraints is defined and applied to different relations to describe which permission(s) are granted to a user based on the role(s) assigned to her. The different types of constraints will be discussed in detail in section 4.

RBAC administrative model

An instance of an RBAC model can include a large number of objects: the complexity and the size of such model instance represent a challenge to manage and maintain it. To ease its management, the RBAC model can be extended with an *administrative* part [38]; this part is shown in figure 1, enclosed with a dashed line. The administrative

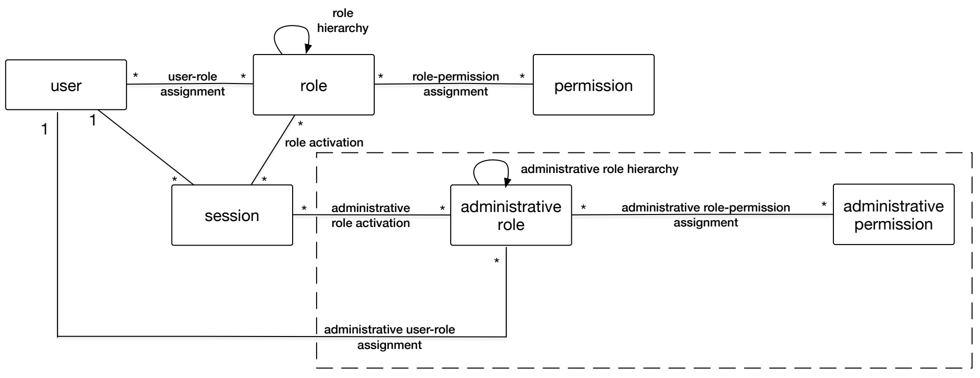

role hierarchy
user
user-role assignment
role
role-permission assignment
permission
1
1
role activation
administrative role hierarchy
session
administrative role activation
administrative role
administrative role-permission assignment
administrative permission
administrative user-role assignment

Figure 1: The original RBAC model [38]; the dashed line encloses the administrative model for RBAC

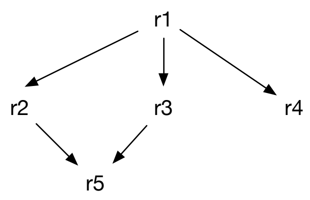


Figure 2: Role hierarchy example

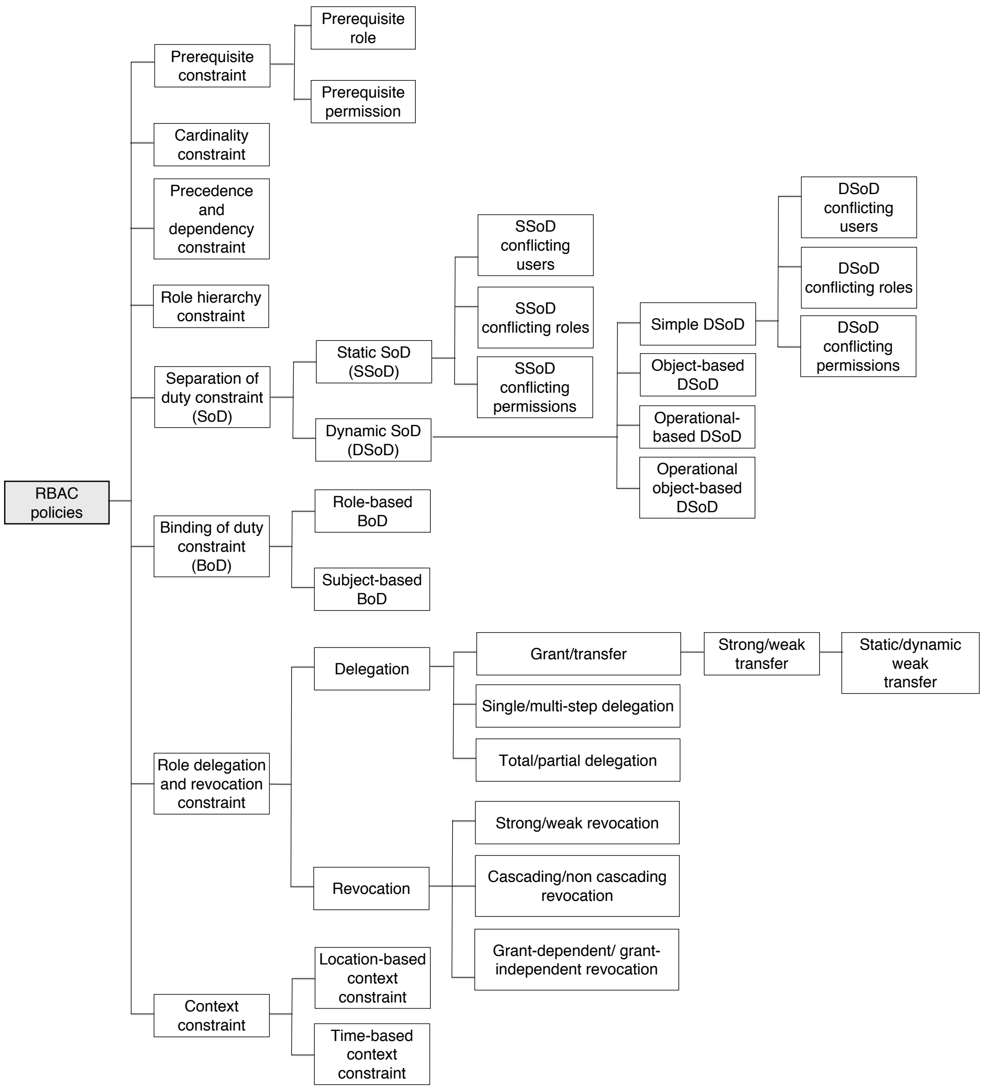
extension contains the concepts of *administrative role* and *administrative permission.* Examples of the latter are assigning a user to a role or adding a new permission or constraint. An *administrative role* can acquire only *administrative permissions* via an *administrative role-permission assignment.* In addition, *administrative user-role assignments* map administrative roles to users. An administrative role hierarchy defines inheritance relations between administrative roles.

4 RBAC policies taxonomy

In this section we present our classification of the existing types of RBAC constraints found in the literature. The taxonomy shown in figure 3, contains at the top level eight types of access control constraints; these are described in detail in the next sub-sections.

4.1 Prerequisite constraint

A prerequisite constraint is a precondition on a role assignment; it can be evaluated either at the role level or at the permission level [38, 4]. A constraint at the role level states that a user can be assigned to a role only if the user has been already assigned to another role. For instance, to acquire the role *developer* in a company, *Bob* must be already an *employee* in the company. A constraint at the permission level indicates that a permission p_1 can be assigned to a role r only if this role already has permission



RBAC policies
Prerequisite constraint
Prerequisite role
Prerequisite permission
Cardinality constraint
Precedence and dependency constraint
Role hierarchy constraint
Separation of duty constraint (SoD)
Static SoD (SSoD)
SSoD conflicting users
SSoD conflicting roles
SSoD conflicting permissions
Dynamic SoD (DSoD)
Simple DSoD
DSoD conflicting users
DSoD conflicting roles
DSoD conflicting permissions
Object-based DSoD
Operational-based DSoD
Operational object-based DSoD
Binding of duty constraint (BoD)
Role-based BoD
Subject-based BoD
Role delegation and revocation constraint
Delegation
Grant/transfer
Strong/weak transfer
Static/dynamic weak transfer
Single/multi-step delegation
Total/partial delegation
Revocation
Strong/weak revocation
Cascading/non cascading revocation
Grant-dependent/ grant-independent revocation
Context constraint
Location-based context constraint
Time-based context constraint

Figure 3: RBAC policies taxonomy

p_2. For example, an employee cannot have the permission *write document* if she does not have the permission *read document.*

4.2 Cardinality constraint

A cardinality constraint can represent a bound on the cardinality of either the *role activation* relation or the *user-role assignment* one [2]. An example of the first is a constraint like: "a user cannot activate more than three roles in a session"; an example of the second is "a user cannot be assigned to more than four roles".

4.3 Precedence and dependency constraint

In some systems, assigning a role to a user does not entail that the user can activate it at anytime. A role that can be activated is called *enabled.* Role enabling and role activation can be controlled by specific constraints that determine precedence/dependency relationships [39] between two or more roles. For example, a policy like "the resident physician role can be enabled only if the attending physician role has been already activated" defines a precedence constraint between the enabling of a role and the activation of another one. A precedence constraint can be complemented with the corresponding dependency constraint on the deactivation of a role; to continue the example above, the corresponding dependency constraint would look like "the attending physician role cannot be deactivated if there is an activated resident physician role".

4.4 Role hierarchy constraint

This type of constraint specifies the assignments of roles through a hierarchy. As explained in section 3, assigning role r to user u implies assigning u all junior roles of r. A role hierarchy constraint can also be applied at the permission level: if a role acquires a permission p, all its sub-roles will also acquire it [38]. For instance, considering figure 2, role r_5 will inherit the permissions of roles r_2 and r_3. The default behavior of this constraint can be overridden by denoting that a role or permission cannot be inherited.

4.5 Separation of duty (SoD) constraint

Separation of duty (SoD) constraints [27] are used to define mutual exclusion relations among rules, permissions, or users. Mutually-exclusive entities are also called *conflicting.* In the literature, there are two types of SoD: static (SSoD) and dynamic (DSoD).

4.5.1 Static separation of duty (SSoD) constraint

This type of separation of duty is also known as strong exclusion [40]. It can refer to users, roles, and permissions [2, 3]. A user-centric static separation, also called *SSoD conflicting users*, states that two conflicting users cannot be assigned to the same role. A role-centric separation, also called *SSoD conflicting roles*, specifies that the same user cannot be assigned to mutually-exclusive roles. SSoD can also be permission-centric: this means that a user is not allowed to acquire two conflicting permissions and, symmetrically, that two conflicting permissions cannot be assigned to the same role.

4.5.2 Dynamic Separation of duty (DSoD) constraint

Dynamic separation of duty deals with user-role activation through a session. In this case, a user is allowed to acquire conflicting roles; however, she cannot activate them at the same time. There are different types of DSoD [40]:

- **Simple DSoD** specifies that conflicting roles cannot be activated in the same session. As in SSoD, simple DSoD can also be user-, role- or permission-centric [4].
- **Object-based DSoD** allows a user to activate two conflicting roles at the same time, as long as she does not operate on the same object. For instance, a user can be an *author* or a *reviewer*; an *author* can submit a paper but cannot be a *reviewer* for it. This type of constraint is also called Resource-based Dynamic Separation of Duty in [26].
- **Operational-based DSoD** aims to prevent a user from performing all the operations in the same business task (e.g., a sequence of operations defined in a workflow). This means that a user can activate two conflicting roles at the same time, as long as the union of the operations allowed by the roles assignment does not correspond to the entire sequence of operations defined in the business task.
- **Operational Object-based DSoD** is a combination of the two previous types of constraint. During the execution of a certain business task, a user can activate two conflicting roles at the same time, even if the union of the operations allowed by the roles assignment correspond to the entire sequence of operations defined in the business task. The only constraint is that no user can perform all the operations on the same object. This type of constraint is also called History-based SoD [40] because the history of the operations performed by a user determines what she is allowed to do. Reference [40] also introduces the concepts of *order-dependent* and *order-independent* history-based SoD. The former requires that a role performs its operations in a particular order; the latter does not take into account the order of operations.

4.6 Binding of duty constraint (BoD)

Unlike SoD constraints, binding of duty constraints define a correlation between a set of permissions; the permissions being correlated and the corresponding operations are also called *bounded.* BoD constraints are usually defined in the context of workflow systems, whose activities can be performed by different subjects with different roles. Reference [42] classifies this type of constraint as *role-based* and *subject-based.* In role-based BoD, the operations allowed by two or more permissions have to be performed by the same role. In subject-based[2] BoD, the same user must perform the operations allowed by the bounded permissions; moreover, the user has to maintain the same role while performing all these operations.

4.7 Role delegation and revocation constraint

A delegation allows a user (called the *delegator)* to transfer the permissions associated with her role (called the *delegated role)* to another user (called the *delegate*). A delegation takes place only if the delegate has not already been assigned to the delegated

[2]The word *subject* refers to a user having activated a certain role.

role or has already received it by means of another delegation. Furthermore, when a hierarchy has been defined for roles, the delegate receives not only the delegated role but also all its sub-roles. A delegation is put to an end through a *revocation* action. In this section, we present the different types of role delegation and revocation constraints which can be set within an RBAC system.

4.7.1 Role delegation constraint

A user can delegate her role or permission to another user. A delegation can be single- or multi-step, total or partial [44], and can be either of type "grant" or "transfer" [11].

A user can acquire a role through a standard user-role assignment (in which case the role is called *original*), or through a delegation (in which case the role is called *delegated*). A *single-step* delegation forbids a user to delegate a delegated role. On the other hand, a *multi-step* delegation allows a user to delegate a delegated (i.e., non-original) role; however, the number of delegation steps is bounded and should not exceed a maximum delegation depth, predefined for the system.

With a *total* delegation, a user delegates all the permissions belonging to a certain role; with a *partial* delegation, a user delegates only a subset of the role permissions.

When a delegation is of type "grant", the delegator can continue to use the role that has been delegated. On the other hand, when the delegation is of type "transfer", right after the delegation the delegator is no longer assigned to the role that has been just delegated.

As mentioned above, when a hierarchy has been defined for roles, the *delegate* receives not only the delegated role but also all its sub-roles. Delegations of type "transfer" can be *strong* or *weak* depending on the assignment of the juniors of the delegated role to the delegator. With a *strong transfer*, the delegator is not assigned to the delegated role and to all its sub-roles anymore. A *weak transfer* can be classified as *static* or *dynamic*. With a *static weak transfer*, the delegator keeps using a subrole r of the delegated role only if she is a member of another senior of role r. In case of a *dynamic weak transfer*, the delegator keeps using a subrole r of the delegated role only if she activates a senior of role r. As an example, consider the role hierarchy in figure 2 and assume that user u_1 is assigned to r_2 and to r_3. If user u_1 delegates her role r_2 to user u_2, the latter will acquire role r_2 and its junior role r_5. If the delegation is a static weak transfer, after the delegation u_1 will still be a member of role r_5, since she is still a member of role r_3, which is a senior of role r_5. On the other hand, if the delegation by user u_1 of role r_2 to user u_2 is of type dynamic weak transfer, the delegator will be still a member of role r_5 after the delegation only if role r_3 is active.

4.7.2 Role revocation constraint

A delegation is often followed by a revocation; in the following we refer to the revocation model proposed in [44].

A role can be revoked either by any user who acquired the role via a user-role assignment, or by the user who delegated the role. In the former case, the revocation is called *grant-independent*; in the latter it is called *grant-dependent.*

The *dominance* of a revocation refers to its effects on the user-role assignment relation, as determined by role hierarchy; it can be either *weak* or *strong.* Consider the case in which a user may be directly assigned to a role or may be assigned to a role by inheriting it through a role hierarchy. A *weak* role revocation only removes the user from the delegated role and does not impact on the other roles acquired through the role hierarchy. A *strong* revocation removes the user from the delegated role and also from the ones inherited through the role hierarchy. For instance, if user u_1 delegates her role r_1 to user u_2, u_2 will acquire r_1 and its juniors r_2, r_3, r_4 and, r_5 as shown in figure 2. With a strong revocation, u_2 will be revoked from r_1 and all its junior roles.

A revocation can affect not only the user who received the role being revoked, but also the other delegate users determined by a multi-step delegation. A *cascading* revocation removes the delegated role assignment and the assignment(s) resulting from a multi-step delegation. A *non-cascading* revocation removes only the requested delegation and does not affect the delegation(s) of the delegated role.

If a delegation has a certain duration, a revocation can be triggered automatically after the duration expires. If a duration is not specified, a delegation remains active until a user revokes it explicitly.

4.8 Context constraint

Context constraints allow a user to perform an action depending on her current location (*location-based context constraints*) [8] or on the time (*time-based context constraints*) at which the action should happen [20].

Contextual information can be assigned to users, roles and/or permissions. A user context refers to her current position and time. A role (respectively, a permission) context refers to the location from and the time at which the corresponding role (respectively, permission) can be activated.

A location can be *physical* or *logical*. A *physical* location corresponds to some specific geographic coordinates; a *logical* location corresponds to a bounded space like a specific room in a building. With a *location-based context constraint*, roles can be enabled and activated when the user's position matches the location specified in the constraint. A role is automatically disabled when the user leaves the geofence determined by the location specified with the constraint.

Time-based context constraints specify the periodicity and/or the duration of role activation [7]. For example, one can constraint a role to be activated only on working days within a predefined range of hours. Moreover, a constraint can limit the cumulative time during which a role is active, e.g., "for three hours per day".

A context-based constraint can be specified at the permission level to prohibit a user to perform a permission assigned to her *active* role, when her contextual information is not valid.

These constraints can also be used to manage conflicting roles that a user can acquire through a role hierarchy. Each of the conflicting roles can have a time-based context constraint that restricts its activation to a certain period of time. If the time windows of the activation of conflicting roles do not overlap with each other, the SoD constraint will not be violated. More in general, context constraints can be applied to a hierarchy constraint to restrict role inheritance and activation depending on location and/or time [34].

5 RBAC model extensions

The original RBAC model, introduced in section 3, was proposed by Sandhu et al. in [38] and is commonly referred to as RBAC96. It is actually defined as a family of reference models: RBAC0, RBAC1, and RBAC2. The basic model, RBAC0, is composed of users, permissions, sessions, and assignment and activation relations.

The other two models are defined incrementally over RBAC0 by adding concepts to it: RBAC1 adds the concept of role hierarchy while RBAC2 adds constraints.

Several researchers extended RBAC96 to support additional type of constraints such as delegation and context. In the rest of this section, we review some of these works.

RDM2000 [44] extends RBAC96 to support role delegation. It includes two types of user-role assignment: in addition to the original user assignment defined in RBAC96, RDM2000 proposes the *delegated user assignment*, which maps a user to a delegated role. A limitation of this work is that it only supports total delegation; this means that a user is not allowed to delegate a subset of her assigned permissions. Another extension related to the concept of delegation is PDM, permission-based delegation authorization model, proposed in [45]. In this model, if a user wants to delegate a set of permissions, she has to create a new role with the required permissions; this role can inherit from any original role. A limitation shared both by RBDM2000 and by PDM is that they only support grant delegation; a complete delegation model supporting both grant and transfer operations has been proposed by Crampton et al. [11]. However, the models provided by RBDM2000, by PDM, and by Crampton et al. do not support the various types of revocation policies presented in section 4. Sohr et al. [22] extends their previous model introduced in [26] to support the role delegation properties of RDM2000; this extension supports the various types of revocation.

Regarding temporal constraints, the first temporal RBAC model, TRBAC, was proposed by Bertino et al. in [7]. TRBAC introduces periodic constraints on role enabling and disabling, which define the period of time (delimited by two time points) during which a role can be enabled/disabled. TRBAC also supports the definition of dependencies among the enable/disable actions of roles; for instance one can require that role r_1 must be enabled whenever role r_2 is disabled. While TRBAC supports temporal constraints only for role enabling/disabling, there is a more general version called GTRBAC [20, 21] that supports temporal constraints also on role activation and assignment. Moreover, GTRBAC adds support for temporal aspects in role hierarchy, cardinality, dependency, and SoD constraints. The GTRBAC was extended in [19] to support delegation by adding the delegation properties supported in PDM.

The first model to support location-based constraint has been GEO-RBAC [8]. It introduces the concept of *spatial role*, which consists of a role and its corresponding region, i.e., the area or the place in which the role can be enabled. In this model, each user is associated to a real position, obtained by a positioning device, and a logical position, which is the mapping of the user's real position into a region of the system. Role enabling is conditioned by the logical position of a user, which should match the one specified in the spatial role definition. This means that the list of enabled roles evolves according to the user's position. An administrative model for GEO-RBAC, called GEO-RBAC Admin was proposed in [12]; in this model an administrative role is defined as a spatial role. While GEO-RBAC supports location-based constraints only for role enabling (and consequently for role activation), the LRBAC model [32] extends these constraints also to user-role and role-permission assignments.

Other works support full context-based constraints by combining spatial and temporal information. The LoTRBAC model [9] extends GTRBAC by assigning a location to each user, role and permission. In this model, location and temporal information control role enabling and activation. Another model, STRBAC [34], allows for defining location- and time-based constraints related to role-to-user and role-to-permission assignments. This may result in a limited subset of permissions allowed for a certain role, at a given time in a given place. Furthermore, STRBAC supports location- and/or

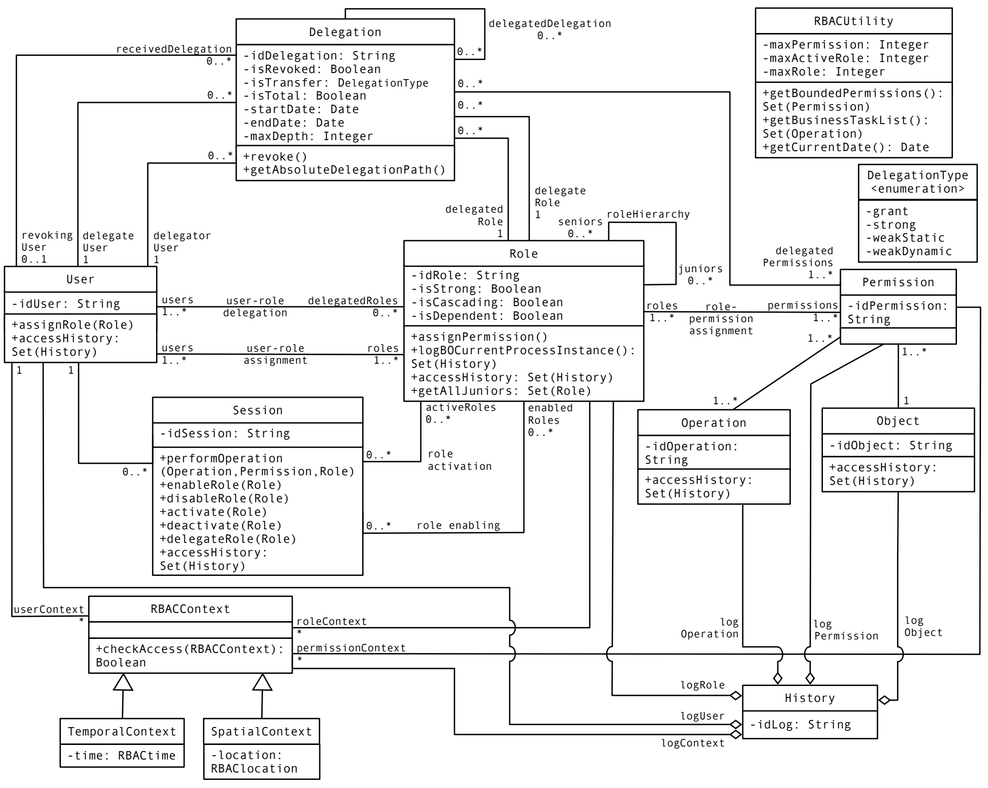


Figure 4: The GemRBAC conceptual model

time-based constraints also for role hierarchy and separation of duty. STRBAC was extended in [35] to support delegation constraints based on contextual information. In STRBAC, a change in a spatio-temporal constraint may lead to the addition of a new role. The GSTRBAC model [31, 1] lifts this requirement by introducing the concept of spatio-temporal zone (*st-zone*). An *st-zone* is an RBAC entity abstracting a location and a time, and is assigned to other entities like users, roles, permissions, and resources. In this mode, a role is enabled if its *st-zone* matches the one of the user; similarly, a permission is enabled if its *st-zone* matches the one of the corresponding resource.

Table 1 shows to which extent the various models discussed in this section support the RBAC constraints included in the taxonomy presented in section 4. As one can notice, there is no extension of the original RBAC model that supports *all* the constraints included in the taxonomy. To overcome this limitation, in the next section we propose a new generalized model for RBAC, which supports all these constraints. This generalized model will constitute the basis for the subsequent formalization with OCL of all the types of constraint identified in the taxonomy.

6 The GemRBAC model

In this section we present our GemRBAC model (shown in figure 4 as a UML class diagram), which extends the original RBAC96 model with additional concepts. The com-

Table 1: Support for RBAC constraints in the various RBAC models

		RBAC96 [38]	RDM2000 [44]	PDM [45]	Crampton et al. [11]	Sohr et al. [26, 22]	TRBAC [7]
Prerequisite	role	+	+	N/A	N/A	+	N/A
	permission	+	+	N/A	N/A	+	N/A
Role hierarchy		+	+	+	+	+	-
Cardinality		+	+	N/A	+	+	+
BoD	role	-	-	-	-	-	-
	subject	-	-	-	-	-	-
Precedence & dependency		-	-	-	-	-	+
SoD	SSoD	+	+	+	+	+	-
	simple DSod	+	+	+	+	+	-
	object DSod	-	-	-	-	+	-
	operational DSod	-	-	-	-	+	-
	history-based DSod	-	-	-	-	+	-
Context	location	-	-	-	-	-	-
	time	-	-	-	-	-	+
Delegation	grant/transfer	-	grant	grant	+	grant	-
	single/multiple	-	+	+	+	+	-
	total/partial	-	total	+	+	total	-
Revocation	dominance	-	-	-	-	+	-
	propagation	-	-	-	-	+	-
	dependency	-	-	-	-	+	-

		GTRBAC [19]	GEO-RBAC [8]	LRBAC [32]	LoTRBAC [9]	STRBAC [34, 35]	GSTRBAC [31, 1]
Prerequisite	role	N/A	N/A	N/A	N/A	N/A	+
	permission	N/A	N/A	N/A	N/A	N/A	+
Role hierarchy		time-based	+	-	+	time and location-based	time and location-based
Cardinality		+	N/A	+	+	N/A	+
BoD	role	-	-	-	-	-	-
	subject	-	-	-	-	-	-
Precedence & dependency		+	-	-	-	-	-
SoD	SSoD	time-based	+	-	+	time and location-based	time and location-based
	simple DSod	time-based	+	-	+	time and location-based	time and location-based
	object DSod	-	-	-	-	-	-
	operational DSod	-	-	-	-	-	-
	history-based DSod	-	-	-	-	-	-
Context	location	-	+	+	+	+	+
	time	+	-	-	+	+	+
Delegation	grant/transfer	grant	-	-	-	grant	-
	single/multiple	+	-	-	-	-	-
	total/partial	+	-	-	-	+	-
Revocation	dominance	-	-	-	-	-	-
	propagation	-	-	-	-	-	-
	dependency	-	-	-	-	-	-

ponents of the original model are modeled as classes (`User`, `Session`, `Role`, `Permission`) and associations (between `User` and `Role`, `User` and `Session`, `Role` and `Permission`). We define a permission as a set of operations that can be performed on an object: we model this by associating the class `Permission` with the classes `Object` and `Operation`. The class `Object` can be extended to include additional information as required by the application needs, e.g., state information for a stateful object.

Role activation is modeled as an association between the `Session` and `Role` classes. We also model role enabling with another association between these two classes.

In GemRBAC the concept of delegation is represented by the class `Delegation`. This class contains various attributes: `startDate` and `endDate` represent the bounds of a delegation period; the boolean attributes `isTotal` and `isRevoked` represent, respectively, whether the delegation is total and whether the delegation has been revoked or not. The attribute `isTransfer` indicates whether the delegation is of type "transfer" or "grant". The concepts of delegator, delegate and revoking users are represented as associations between the `User` and `Delegation` classes. Similarly, the concepts of the role of the delegator, the role of the delegate, and the delegated role are represented as an association between class `Role` and class `Delegation`. We also model the set of delegated roles as an association between classes `User` and `Role`. The specific type of revocation is represented by specific boolean attributes of the `Role` class: `isDependent`, `isStrong`, and `isCascading`.

The context is modeled with the class `RBACContext`, which has two subclasses, `TemporalContext` and `SpatialContext`. The `RBACContext` class represents spatial and temporal information, which are associated with each instance of the `User`, `Role` and `Permission` classes. The temporal context refers to the current time or the time on which a given role, respectively permission, can be enabled/assigned. The spatial context refers to a specific bounded area or geographical location. It can be assigned to a user, role or permission. A role, respectively permission, is enabled/assigned if its location matches the user's position.

Since policies such as *History-based SoD* require a record of operations performed over time, we introduce the `History` class. An instance of this class records that a user performed a certain operation on a given object according to a given permission, in a certain location, at a certain time, while having a certain role; these data are gathered through the associations with (respectively) `User`, `Operation`, `Object`, `Permission`, `TemporalContext`, `SpatialContext`, and `Role`.

7 OCL specification of RBAC policies

In this section we show how the RBAC policies described in section 4 can be formalized as OCL constraints on the GemRBAC model. This formalization aims to precisely specify the semantics of such policies in such a way that they can be operationalized, for example through an OCL constraint checker. For the purpose of the formalization, we enrich the model with some helper classes and operations, also included in figure 4.

The Ecore version of the GemRBAC model, the OCL constraints defined in this section, and the model instances that violate/satisfy them are available at `https://github.com/AmeniBF/GemRBAC-model`.

7.1 Prerequisite constraint

The prerequisite constraint can specify a pre-condition either for user-role assignment or for role-permission assignment. In the first case, the constraint states that to acquire role r_1, the user must have been already assigned to role r_2. This constraint can be written in OCL as a pre-condition of the operation `assignRole`:

```
1 context User::assignRole(r:Role):
2 pre PreqRole:
3 let r2:Role = Role.allInstances() -> select (r:Role | r.idRole ='r2') ->
4               any(true),
5     roleSet: Set(Role) = self.roles -> union(self.delegatedRoles) -> asSet()
6 in  r.idRole = 'r1' implies  roleSet -> includes(r2)
```

In this constraint we first select (line 4) the instance[3] of role r_2 from all the instances of class `Role`. In line 6, the implication states that if the role being assigned (parameter `r` of the operation `assignRole`) is r_1 then r_2 must be among the roles already assigned or delegated to the user; these roles are derived from navigating the `roles` and `delegatedRoles` associations.

The prerequisite constraint on role-permission assignment has a similar structure:

```
context Role:: assignPermission(p:Permission):
pre PreqPermisssion:
let p2:Permission = Permission.allInstances() -> select (p:Permission |
                    p.idPermission = 'p2') -> any(true)
in  p.idPermission = 'p1' implies self.permission -> includes(p2)
```

7.2 Cardinality constraint

A cardinality constraint on the role activation relation is expressed in OCL as an invariant of the class `Session`:

```
context Session inv Cardinality:
let u: RBACUtility = RBACUtility.allInstances() -> any(true)
in  self.activeRoles -> size() <= u.maxActiveRole
```

In this expression, the number of roles activated for the current session is determined by the cardinality of the `activeRoles` association between classes `Session` and `Role`; `maxActiveRole` is a constant defined in the class `RBACUtility`.

The constraints for the number of roles assigned to a user and for the number of permissions assigned to a role are defined in a similar way for classes `User` and `Role`:

```
context User inv Cardinality:
let u: RBACUtility = RBACUtility.allInstances() -> any(true),
    roleSet : Set(Role) = self.roles -> union(self.delegatedRoles)-> asSet()
in  self.roleSet -> size() <= u.maxRole

context Role inv Cardinality:
let u: RBACUtility = RBACUtility.allInstances() -> any(true)
in  self.permissions -> size() <= u.maxPermission
```

Notice that when expressing the constraint on the number of roles assigned to a user, we consider both assigned and delegated roles.

[3] Although the operation **`allInstances()`** returns all the instances of a class, we assume that each role in the system corresponds to exactly one specific instance of class `Role`.

7.3 Precedence and dependency constraint

Precedence and dependency constraints control the enabling and the deactivation of roles. A precedence constraint on the enabling of a role with respect to the activation of another one can be expressed as a pre-condition of the operation `enableRole`:

```
1 context Session::enableRole(r:Role):
2 pre RoleEnablingPrecedence:
3 let r2: Role = Role.allInstances() -> select (a: Role | a.idRole  = ’r2’) ->
4               any(true)
5 in  r.idRole = ’r1’ implies Session.allInstances() ->
6                           exists (s: Session | s.activeRoles -> includes(r2))
```

In the OCL expression above, we first select the instance corresponding to role r_2 (line 4), and then the implication at line 5 states that if the role being enabled (parameter `r` of the operation `enableRole`) is r_1 then r_2 must be among the activated roles; the list of these roles is derived from the `activeRoles` association. The corresponding dependency constraint can be expressed in a similar way:

```
context Session::deactivateRole(r:Role):
pre RoleActivationDependency:
let r2:Role = Role.allInstances() -> forAll(a: Role | a.idRole = ’r2’)->
             any(true)
in r.idRole = ’r1’ implies (Session.allInstances() ->
                           exists (s:Session | s.activeRoles -> excludes(r2)))
```

7.4 Role hierarchy constraint

A role hierarchy constraint can be expressed on user-role and permission-role assignment relations. In the first case, it states that if a user acquires a role, she will also acquire all its juniors. This can be expressed in OCL as a post-condition of the operation `assignRole` as follows:

```
context User::assignRole(r:Role):
post RoleHierarchy:
self.roles -> includesAll(r.juniors)
```

In this expression the roles assigned to the user, derived from the `roles` association, should include all junior roles of the role being assigned (parameter `r` of operation `assignRole`); the junior roles are derived from the `r.juniors` association.

A role hierarchy constraint on the role-permission assignment states that if a role acquires a permission, all its sub-roles will acquire it. This constraint is defined as a post-condition of the operation `assignPermission`:

```
context Role::assignPermission(p:Permission):
post RoleHierarchy:
self.juniors -> forAll (r: Role | r.permissions -> includes(p))
```

In this expression, we check if each sub-role is associated to the permission being assigned (parameter `p` of the operation `assignPermission`).

7.5 Separation of duty constraint (SoD)

7.5.1 Static SoD

The static SoD (SSoD) can be user-, role- or permission-centric. The user-centric SoS specifies that role r_1 can be assigned either to $user_1$ or to $user_2$, but not to both. This constraint can be expressed in OCL as an invariant of the class `Role`:

```
 1 context Role inv SSoDCU:
 2 if self.idRole =’r1’ then
 3    let u1:User = User.allInstances() -> select (u:User | u.idUser=’u1’) ->
 4                 any(true),
 5        u2:User = User.allInstances() -> select (u:User | u.idUser=’u2’) ->
 6                 any(true)
 7    in if self.users -> includes(u2) or self.users -> includes(u1) then
 8          self.users -> includes(u2) xor self.users -> includes(u1)
 9       endif
10 endif
```

In the OCL expression above, we first select the instances of users $user_1$ (line 4) and $user_2$ (line 6). In line 7, we state that the users assigned to the role should contain either $user_1$ or $user_2$, but not both (exclusive OR); these users are derived from the `users` association.

The role-centric and permission-centric SoD are defined in a similar way as invariants of the `User` and `Permission` classes, respectively:

```
context User inv SSoDCR:
let r1:Role = Role.allInstances() -> select (r:Role | r.idRole=’r1’) ->
              any(true),
    r2:Role = Role.allInstances() -> select (r:Role | r.idRole=’r2’) ->
              any(true),
    roleSet : Set(Role) = self.roles -> union(self.delegatedRoles) -> asSet()
in  if roleSet -> includes(r2) or self.roles -> includes(r1) then
        roleSet -> includes(r2) xor self.roles -> includes(r1)
     endif

context Role inv SSoDCP1:
let p1:Permission = Permission.allInstances()-> select (p:Permission |
                    p.idPermission=’p1’) -> any(true),
    p2:Permission = Permission.allInstances()-> select (p:Permission |
                    p.idPermission=’p2’) -> any(true)
 in if self.permissions -> includes(p2) or self.permissions -> includes(p1)
     then
       self.permissions -> includes(p2) xor self.permissions -> includes(p1)
     endif

context Permission inv SSoDCP2:
if self.idPermission  =’p1’ then
   let r1:Role = Role.allInstances() -> select (r:Role | r.idRole=’r1’) ->
                 any(true),
       r2:Role = Role.allInstances() -> select (r:Role | r.idRole=’r2’) ->
                 any(true)
   in if self.roles -> includes(r2) or self.roles -> includes(r1) then
         self.roles -> includes(r2) xor self.roles ->  includes(r1)
      endif
endif
```

7.5.2 Dynamic SoD

Unlike static SoD, dynamic SoD (DSoD) allows a user to acquire two conflicting roles but she cannot activate them at the same time. To express that roles r_1 and r_2 should not be active in the same session, we can write the following OCL constraint as an invariant of the class `Session`:

```
context Session inv DSoD:
let  r1:Role = Role.allInstances() -> select (r:Role | r.idRole='r1') ->
               any(true),
     r2:Role = Role.allInstances() -> select (r:Role | r.idRole='r2') ->
               any(true)
in   if self.activeRoles -> includes(r2) or self.activeRoles -> includes(r1)
    then
        self.activeRoles -> includes(r2) xor self.activeRoles -> includes(r1)
     endif
```

Object-based SoD is another variation of DSoD which allows a user to activate two conflicting roles at the same time, as long as she does not operate on the same object. It can be expressed in OCL as a pre-condition of the operation `performOperation` of the `Session` class:

```
 1 context Session::performOperation(op:Operation, p:Permission, r:Role):
 2 pre ObjectDSOD:
 3 let r2:Role = Role.allInstances() -> select (r:Role | r.idRole='r2') ->
 4               any(true),
 5     r1:Role = Role.allInstances() -> select (r:Role | r.idRole='r1') ->
 6               any(true),
 7     logr2: Set (History) = self.user.accessHistory() -> select (a: History |
 8               a.role= r2),
 9     logr1: Set (History) = self.user.accessHistory() -> select (a: History |
10               a.role= r1)
11 in  if r = r1 then
12        (logr2 -> select (a: History| a.object= p.object)) -> isEmpty()
13     else if r = r2 then
14            (logr1 -> select (a: History| a.object= p.object)) -> isEmpty()
15          endif
16     endif
```

In this constraint we refer to instances of the `History` class, which keeps track of each operation performed in the system, recording the user who performed it, the role she had, the object on which the operation was performed, the time and the location. We use the operation `accessHistory` of the class `User` to retrieve the instances of `History` filtered on the two conflicting roles (lines 8 and 10). For each role that the input parameter `r` can assume (in this case r_1 or r_2, lines 11 and 13), we check (lines 12 and 14) whether the history of the conflicting role(s) does not contain any operation performed on the same object as the one specified in the input parameter `p`.

With an Operational-based DSoD constraint, a user can activate two conflicting roles at the same time, as long as the union of the operations allowed by the roles assignment does not correspond to the entire sequence of operations defined in a business task. This can be expressed in OCL as an invariant of the class `Session`:

```
 1 context Session inv OperationalDSoD:
 2 let r1:Role = Role.allInstances() -> select (r:Role | r.idRole='r1') ->
 3              any(true),
 4     r2:Role = Role.allInstances() -> select (r:Role | r.idRole='r2') ->
 5              any(true),
 6     u: RBACUtility = RBACUtility.allInstances() -> any(true),
 7     opBT: Set(Operation) = u.getBusinessTaskList(),
 8     op:Set(Operation) = r1.permissions.operations -> asSet() ->
 9                         union(r2.permissions.operations -> asSet())
10 in (self.activeRoles -> includes (r1) and self.activeRoles -> includes (r2))
11    implies (opBT - op) -> notEmpty()
```

On line 7 we retrieve the list of operations defined in the business task by calling the `getBusinessTaskList` operation of the class `RBACUtility`. We have to check that the union of the operations allowed by the two conflicting roles is a proper subset of the business task operations. This is equivalent to stating that the difference between the two sets is not empty (line 11). This check is done if both roles are active.

The History-based DSoD constraint combines both the Object-based one and the Operational-based one. Differently from these two, History-based DSoD allows a user to activate two conflicting roles at the same time, as long as the user does not perform all the operations on the same object. This can be specified as a pre-condition of the operation `performOperation` of the `Session` class:

```
 1 context Session::performOperation(op:Operation, p:Permission, r:Role):
 2 pre HistoryDSOD:
 3 let u: RBACUtility = RBACUtility.allInstances() -> any(true),
 4     opBT: Set(Operation) = u.getBusinessTaskList(),
 5     r1:Role = Role.allInstances()-> select (r:Role | r.idRole='r1') ->
 6              any(true),
 7     r2:Role = Role.allInstances()-> select (r:Role | r.idRole='r2') ->
 8              any(true),
 9     perm1: Set (Permission)= r1.permissions-> select (a |
10                                a.object = p.object),
11     perm2: Set (Permission)= r2.permissions-> select (a |
12                                a.object = p.object),
13     opObjBT: Set(Operation) = (perm1.operations -> union(perm2.operations))->
14                                select (op:Operation | opBT ->
15                                includes (op)) ->asSet(),
16     logr2: Set (History)=  self.user.accessHistory() ->
17                                select (a: History  | a.role = r2
18                                and perm2 -> includes (a.permission)),
19     logr1: Set (History)=  self.user.accessHistory() ->
20                                select (a: History | a.role = r1
21                                and  perm1-> includes (a.permission)),
22     log: Set (History) = logr1 -> union(logr2),
23     opLog: Set (Operation)= log -> collect (l | l.operation) -> asSet(),
24     newopLog:Set (Operation) = opLog -> including (op)
25 in (opObjBT - newopLog)-> notEmpty()
```

In the OCL expression above, we first retrieve the list of operations defined in the business task. Then, we select the conflicting roles r_1 and r_2 (lines 6 and 8), and then we compute (line 15) the list of operations `opObjBT` belonging to the business task and that can be performed on the input object (`p.object`) according to the permissions of the two roles. Afterwards, we compute the list `opLog` of operations performed by the user (either under role r_1 or r_2) on the input object (line 22). The pre-condition then checks whether the set resulting from adding the input parameter `op` to `opLog` is still a proper subset of `opObjBT`.

7.6 Binding of duty constraint (BoD)

Binding of duty constraints define a correlation between a set of permissions. A role-based BoD constraint requires that bounded operations must be executed by the same role. This constraint can be specified in OCL as a pre-condition of the operation `performOperation` of the `Session` class:

```
1  context Session::performOperation(op:Operation, p:Permission, r:Role)
2  pre RoleBoD:
3  let u: RBACUtility = RBACUtility.allInstances() -> any(true),
4      boundedPermissions: Set (Permission)= u.getBoundedPermissions(),
5      roles: Set (Role)= Role.allInstances() -> select (r:Role |
6                            r.permissions -> includesAll(boundedPermissions)),
7      roleLog: Set (History)=  r.logBOCurrentProcessInstance()
8  in if boundedPermissions -> includes(p) and roleLog -> isEmpty() then
9     roles -> forAll (r:Role |r.logBOCurrentProcessInstance() -> isEmpty())
10    endif
```

In the OCL expression above, we first retrieve the set of bounded permissions and their assigned roles (lines 4 and 6). Then, we determine which bounded operations have been performed by role `r` in the current process instance[4] by calling the operation `logBOCurrentProcessInstance()` of the `Role` class (line 7). If the operation being performed corresponds to a bounded operation and none of the bounded operations have been performed in the current process instance by role `r` (line 8), then the pre-condition is satisfied if none of the bounded operations have been previously performed by *any* other role different from `r` (line 9). In other words, if a user with role `r` has already performed a bounded operation (corresponding to the case in which the condition `roleLog->isEmpty()` is false), any other user with the *same role* is allowed to perform another bounded operation in the current process instance.

The subject-based BoD constraint requires that bounded operations must be executed by the same subject (user and role). This constraint can be expressed in OCL as:

```
context Session::performOperation(op:Operation, p:Permission, r:Role)
pre SubjectBoD:
```

[4]We recall that BoD constraints are usually defined in the context of process-based workflow systems.

```
let u: RBACUtility = RBACUtility.allInstances() -> any(true),
    boundedPermissions: Set (Permission)= u.getBoundedPermission(),
    roles: Set (Role)= Role.allInstances() -> select (r:Role |
                      r.permissions-> includesAll(boundedPermissions)),
    subjectLog: Set (History)= r.logBOCurrentProcessInstance() ->
                      select (a: History | a.user= self.user)
in if boundedPermissions -> includes (p) and subjectLog -> isEmpty() then
      roles -> forAll (r:Role | r.logBOCurrentProcessInstance() -> isEmpty()
endif
```

This OCL constraint is similar to the one defined for role-based BoD. However, the log `subjectLog` corresponds to the bounded operations that have been performed by the user `user` with role `r` in the current process instance.

7.7 Role revocation and delegation constraints

7.7.1 Role delegation constraint

A delegation is characterized by a delegated role, a delegator, a delegate and their corresponding roles. In a multi-step delegation, a user is allowed to delegate a delegated role according to a maximum delegation depth (hereafter called *maxDepth*). This type of delegation can be specified in OCL as an invariant of the class `Delegation`

```
context Delegation inv MultiStepDelegation:
self.getAbsoluteDelegationPath() -> size() <= self.maxDepth
```

In the OCL expression shown above, the operation `getAbsoluteDelegationPath` returns the list of delegation steps starting from the original (non-delegated role). The size of this list is then compared with the attribute `maxDepth`. The single-step delegation constraint can be defined as a multi-step delegation with a maximum delegation depth equal to 1.

A delegation can be total or partial depending on the number of permissions being delegated. A total delegation delegates all the permissions belonging to a certain role; it can be specified in OCL as an invariant of the `Delegation` class:

```
context Delegation inv TotalDelegation:
self.isTotal implies
self.delegatedPermissions = self.delegatedRole.permissions
```

This expression states that if the delegation is total (represented by the attribute `isTotal`) then the list of delegated permissions (derived from the association `delegatedPermissions`) should be equal to the list of permissions associated to the delegated role.

A partial delegation (characterized by the attribute `isTotal` being false) is defined in a similar way:

```
context delegation inv PartialDelegation:
not (self.isTotal) implies
(self.delegatedRole.permissions - self.delegatedPermissions) -> notEmpty()
```

A delegation can be either of type “grant” or “transfer”. While a “grant” type delegation does not affect the permissions of the delegator, in case of a delegation of type “transfer”, the delegator cannot use the delegated role after the delegation. A delegation of type “transfer” can be either *strong* or *weak*. In case of *strong transfer*, in addition to the delegated role, the delegator is no longer assigned to any of its juniors. This constraint can be expressed in OCL as an invariant of the `Delegation` class:

```
1  context Delegation inv StrongTransfer:
2  let roles: Set(Role) = self.delegatedRole.getAlljuniors() -> including (self.
       delegatedRole)
3  in self.isTransfer = delegationType::strong implies
4  self.delegatorUser.roles -> excludesAll(roles)
5                              and self.delegateUser.delegatedRoles ->
6                              includes(self.delegatedRole)
```

In the OCL expression shown above, we first retrieve the list of roles from the `Delegation` object, including the delegated role and its juniors (line 2). The operation `getAlljuniors` returns the juniors of the delegated role, walking through the transitive closure of the hierarchy relation. For instance, the operation `getAlljuniors` applied to role r_1 in figure 2 returns the list of roles: r_2, r_3, r_4 and r_5. The OCL expression at line 3 states that if the delegation is of type *strong transfer* (represented by the expression `self.isTransfer = delegationType::strong`) then the list of roles assigned to the delegator (derived from the `delegatorUser.roles` association) should include neither the delegated role nor any of its juniors. Besides, the list of roles delegated to the delegate should include the delegated role; these roles are derived by navigating the `delegateUser.delegatedRoles` association.

A delegation of type *weak transfer* can be either *static* or *dynamic*. In case of *static weak transfer*, the delegator keeps using a subrole r of the delegated role only if she is a member of another senior of role r. This constraint can be expressed in OCL as an invariant of the `Delegation` class:

```
1  context Delegation inv StaticWeakTransfer:
2  let acquiredRoles: Set(Role) = self.delegatorUser.roles ->
3                  union(self.delegatorUser.delegatedRoles),
4      allowedRoles: Set(Role) = self.delegatedRole.getAllJuniors() ->
5                  select (r : Role | (r.seniors -> excluding(delegatedRole)) ->
6                  exists (r1 : Role | acquiredRoles ->includes(r1))),
7      roles: Set(Role) = (self.delegatedRole.getAllJuniors() ->
8                  including(self.delegatedRole)) - allowedRoles
9  in self.isTransfer = delegationType:: weakStatic implies
10         self.delegatorUser.roles -> excludesAll(roles)
11         and  self.delegatorUser.roles -> includesAll(allowedRoles)
12         and self.delegateUser.delegatedRoles -> includes(self.delegatedRole)
```

In the OCL expression shown above, we first retrieve the list (`acquiredRoles`) of roles available to the delegator, defined as the union of the roles assigned to and delegated to the delegator (lines 2 and 3). Then, we select among the subroles of the delegated role, the roles (`allowedRoles`) that the delegator is allowed to acquire. We check if one of the juniors of the delegated role has another senior in the list `acquiredRoles` (lines 4–6). We compute the list of roles that the delegator cannot use after the transfer, keeping into account the `allowedRoles` (lines 7 and 8). If the transfer is of type *static weak* (condition checked at line 9) the list of roles assigned to the delegator should include neither the delegated role nor any of its juniors (except for those allowed by the hierarchy relation). Finally, the list of roles delegated to the delegate should include the delegated role (line 12).

The constraint for the delegation of type *dynamic weak transfer* has a similar structure:

```
 1 context Delegation inv DynamicWeakTransfer:
 2 let  acquiredRoles : Set(Role) = self.delegatorUser.roles ->
 3                               union(self.delegatorUser.delegatedRoles),
 4      allowedRoles : Set(Role) = self.delegatedRole.getAllJuniors() ->
 5                              select (r : Role | (r.seniors ->
 6                              excluding(delegatedRole)) -> exists(r1 : Role |
 7                              self.delegatorUser.sessions -> exists(s:Session |
 8                              s.activeRoles -> includes(r1)))),
 9      roles : Set(Role) = (self.delegatedRole.getAllJuniors() ->
10                              including(self.delegatedRole)) - allowedRoles
11 in  self.isTransfer = delegationType:: weakDynamic implies
12         self.delegatorUser.roles -> excludesAll(roles)
13         and self.delegatorUser.roles -> includesAll(allowedRoles)
14         and self.delegateUser.delegatedRoles -> includes(self.delegatedRole)
```

Notice that in this case the list `allowedRoles` (lines 5–8) includes the subroles having an active senior in the delegator session.

7.7.2 Role revocation constraints

A delegation is revoked by setting the attribute `isRevoked` to true. The delegation is revoked automatically when its duration expires; this constraint can be specified as an invariant of the `Delegation` class as follows:

```
context Delegation inv AutomaticRevocation:
let u: RBACUtility = RBACUtility.allInstances() -> any(true),
in  u.currentDate >= self.endDate implies self.isRevoked
```

A revocation is called grant-dependent if only the delegator is allowed to revoke the delegation. On the other hand, a grant-independent revocation allows not only the delegator but also any *original* user to revoke the delegation. This constraint can be expressed in OCL as a pre-condition of the operation `revoke` of the `Delegation` class:

```
1 context Delegation::revoke()
2 pre RevacationDependency:
3 if self.delegatedRole.isDependent then
4      self.revokingUser = self.delegatorUser
5 else
6      self.revokingUser = self.delegatorUser or self.delegatedRole.users ->
7                                               includes(self.revokingUser)
8 endif
```

In the OCL expression above, we first check if the revocation is grant-dependent, by checking the attribute `isDependent` of the association `delegatedRole` (line 3). If this is the case, only the delegator is allowed to revoke the delegation (line 4). Otherwise, the delegation can be revoked either by the delegator or by any user *assigned* to the delegated role (lines 6 and 7).

A revocation can be classified as strong or weak according to its dominance. A strong revocation removes from a user not only the delegated role but also its junior roles. A strong revocation can be expressed in OCL as a post-condition of the operation `revoke` of the `Role` class:

```
context Delegation::revoke()
post StrongRevocation:
self.delegatedRole.isStrong implies
self.delegateUser.delegatedRoles -> excludesAll(self.delegatedRole.
    getAllJuniors())
```

In the constraint above, the implication states that if the revocation is strong (as determined by the attribute `isStrong`), the set of roles received by the delegation (`delegateUser.delegatedRoles`) should not include juniors of the delegated role.

A revocation can be classified as cascading or non-cascading according to its propagation. A cascading revocation removes all delegations resulting from a multi-step delegation. It can be specified in OCL as a post-condition of the operation `revoke` of the `Delegation` class:

```
context Delegation::revoke():
post CascadingRevocation:
self.delegatedRole.isCascading implies
self.delegatedDelegation -> forAll (d: Delegation | d.isRevoked = true)
```

In the constraint above, the implication states that in case of a cascading revocation (as determined by the attribute `isCascading`), all the delegated delegations should be revoked; this list of delegations is derived from the association `delegatedDelegation`.

7.8 Context constraint

A location-based constraint states that the role should be enabled (or assigned) if the user is located in the location assigned to the role itself. Moreover, the role should be disabled (or unassigned) when the user is not located anymore in the location assigned to the role. The location-based constraint on role enabling can be specified in OCL as an invariant of the class `Session` as follows:

```
context Session inv locationbased:
let userLocation: SpatialContext = self.user.userContext.
                           oclAsType(SpatialContext),
    disabled: Set (Role) = self.user.roles -> select (r:Role |
                           r.roleContext.oclAsType(SpatialContext) ->
                           forAll (loc: SpatialContext |
                           not(loc.checkAccess(userLocation)))),
    enabled: Set (Role) = self.user.roles -> select (r:Role |
                           r.roleContext.oclAsType(SpatialContext) ->
                           exists (loc: SpatialContext |
                           loc.checkAccess(userLocation) = true))
in self.enabledRoles -> excludesAll(disabled) and self.enabledRoles ->
    includesAll(enabled)
```

In the OCL expression above, we first select among the set of roles assigned to the user in the current session the roles that should be disabled. For each role, we check (by means of the `checkAccess` operation of the class `RBACContext`) whether the role's spatial context matches the user's location; if this is not the case, the role should be disabled (lines 4–7). In a similar way, we retrieve the list of roles that should be enabled. If any location in the role's spatial context matches the user's location, the role is enabled (lines 8–11). Then we check whether the roles in the two lists belong or not to the list of roles enabled in the current session (line 12).

A time-based constraint can be expressed in a similar way by replacing the instances of `SpatialContext` with instances of `TemporalContext`. More in general, a context-based constraint, combining both time and location, can be expressed in OCL as an invariant of the class `Session`:

```
context Session inv RoleContext:
let userContext: RBACContext = self.user.userContext,
    disabled: Set (Role) = self.user.roles -> select (r:Role |
                              r.roleContext -> forAll (c: RBACContext |
                              not(c.checkAccess(userContext)))),
    enabled: Set (Role) = self.user.roles -> select (r:Role |
                             r.roleContext -> exists (c: RBACContext |
                             c.checkAccess(userContext) = true))
in self.enabledRoles -> excludesAll(disabled) and self.enabledRoles ->
    includesAll(enabled)
```

A context-based constraint defined at the permission level can be specified in OCL as a pre-condition of the `performOperation` of the class `Session`:

```
context Session::performOperation (op: Operation, p:Permission, r:Role)
pre PermissionContext:
let userContext: RBACContext = self.user.userContext
in  p.permissionContext -> select(c:RBACContext | c.checkAccess(userContext))
     -> notEmpty()
```

In the OCL expression above, we check whether there is at least one instance of the contextual information assigned to the permission `p` that matches the user's context.

8 Application

The GemRBAC model and the OCL formalization of the various types of RBAC constraints presented in the previous sections represent our main contribution towards an integrated framework for the definition of the RBAC constraints identified in our taxonomy (see section 4). Our goal is to enable the specification of policies that make use of the concepts previously proposed in the literature, using a unified model (GemRBAC) and a standardized language (OCL) for their definition. With respect to the state of the art, not only we consider all types of RBAC policies proposed in the literature, but we also use a common model and notation to define them, improving their understanding.

In terms of application, from the point of view of the definition of RBAC policies, we believe that this work can represent a one stop source for security engineers, who can access the taxonomy of the various types of RBAC constraints, determine their exact meaning by referring to their formalization as OCL constraints on the GemRBAC model, select the constraints that suit the needs of their organization, and operationalize them based on readily-available OCL checkers.

We also maintain that our framework can be a basis on which to further develop a model-driven approach for the verification of RBAC policies. Verification should

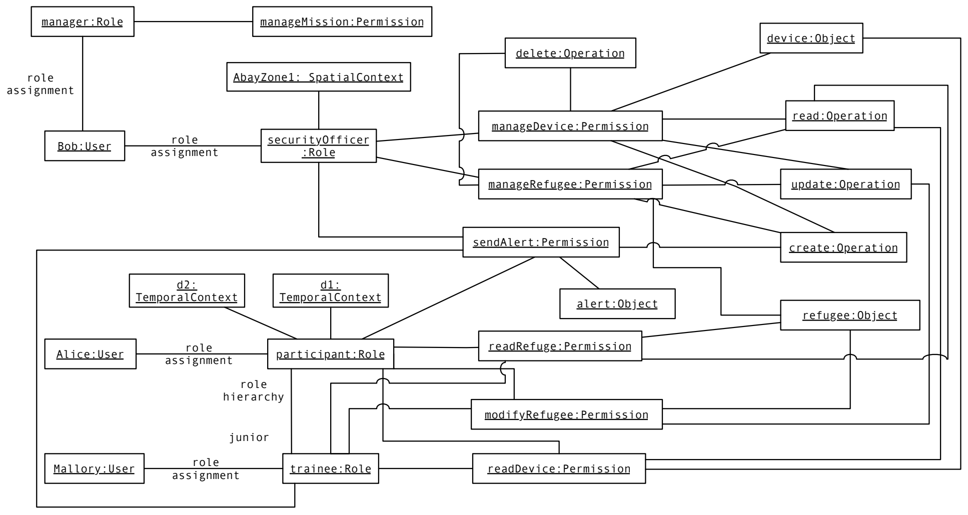


Figure 5: Initial system state

be intended here in its broadest scope, including design-time verification (e.g., consistency checking of policies [17, 18]) and run-time enforcement (e.g., allowing access to resources only if an instance of the GemRBAC model satisfies the constraints associated with it). In particular, for the latter, we assume that the run-time infrastructure collects snapshots representing the state of the system (from the point of view of RBAC) as instances of the GemRBAC model. RBAC policies can be defined as OCL constraints that the instances of the GemRBAC model should satisfy; these constraints are based on the OCL templates proposed in section 7. An OCL checker (such as Eclipse OCL [15]) can be used to check if a model instance satisfies the OCL constraints associated with it, resulting in the enforcement of the corresponding access control policies.

Although the definition of approaches for the verification of RBAC constraints (including both consistency checking and model-driven run-time enforcement) is out of the scope of this paper, in the rest of this section we describe an example scenario, showing how the GemRBAC model can be instantiated in some states corresponding to run-time changes of the system, and how we can define RBAC constraints using the OCL templates proposed in the previous section.

Example Scenario

This example scenario is inspired by a real-world application developed by our partner HITEC Luxembourg, which develops situational-aware systems for emergency scenarios. The application is an integrated communication solution, to be used in case of an emergency scenario (e.g., a natural large-scale disaster or a civil war situation), to ease the process of assisting refugees and/or casualties in such situations.

The application allows different (humanitarian) organizations to participate to various missions. During a mission, a user of the application can send alerts to request treatment services for injured people. Each user belongs to at least one organization and can be assigned to one or many missions. Each mission is characterized by a name, a start date, an end date, and a geofence. The latter is a geographic boundary that defines where users assigned to a certain role should be situated during the mission period.

The membership of a user to an organization or to a mission does not automatically grant the access to the corresponding resources. Following the principles of RBAC, the access is allowed (or denied) according to the user's role. In the rest of this section, we consider the mission *Philippine*, which starts on date *d1*, ends on date *d2*, and is situated within the geofence labeled *AbayZone1*. We refer to the following RBAC entities:

- **users** = {Bob, Alice, Mallory};
- **roles** = {securityOfficer, participant, trainee, manager};
- **permissions** = {readRefugee, updateRefugee, manageRefugee, readDevice, manageDevice, sendAlert, manageMisssion};
- **operations** = {create, read, update, delete};
- **objects** = {refugee, device, alert};
- **spatial context** = {AbayZone1};
- **temporal context** = {d1, d2}.

Figure 5 depicts an instance of the GemRBAC model representing the initial system state for the mission *Philippine*. Each role is assigned to a set of permissions. A permission is an abstraction of an object and a set of operations. For instance, the permission *manageDevice* corresponds to the execution of the the operations *create*, *read*, *update*, and *delete* on the object *device*. All roles are given the permission *sendAlert*, to send an alert. The role *securityOfficer* can manage all the resources; permissions of the form *manage** include all the operations defined in the system. Role *participant* is entitled the permission *updateRefugee*, which allows to update the details of an existing refugee. Moreover, role *participant* is given the permissions of the form *read** for the objects of type *device* and *vehicle*. Role *trainee* is a subrole of role *participant*; hence, it inherits all the permission of its senior. Role *manager* can perform administrative operations on the mission through the permission *manageMission*. Roles are assigned to users as follows: *Bob* is assigned to role *securityOfficer* and to role *manager*, *Alice* is assigned to role *participant*, and *Mallory* is assigned to role *trainee*. In addition to the user-role and role-permission assignments shown in figure 5, some additional constraints can further restrict the user access. We define the following constraints:

C1: *role* participant *is enabled for the entire duration of the mission.*

C2: *role* securityOfficer *is enabled if the user is situated in the mission geofence.*

C3: *role* trainee *is enabled only if role* securityOfficer *is active.*

C4: *right after a delegation of type strong transfer, the delegator is no longer assigned to the delegated role and all its juniors.*

The time-based constraint **C1** restricts the role activation; no user can activate role *participant* before the starting of the mission. This constraint can be expressed using the template (*locationbased*) provided on page 25, by replacing the instances of

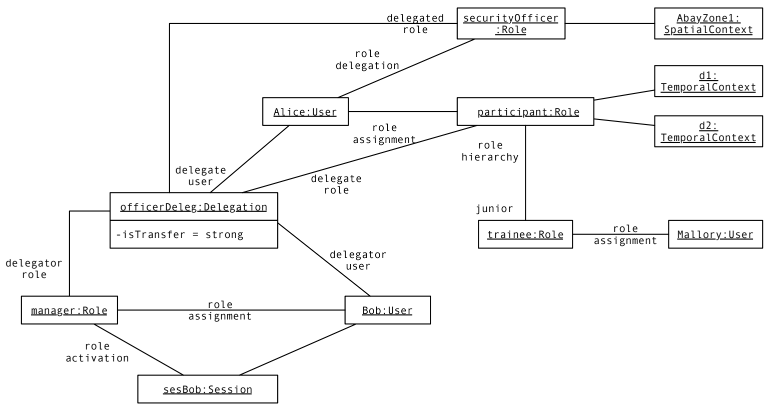


Figure 6: System state after the delegation of role *securityOfficer*

`SpatialContext` with instances of `TemporalContext`. At the beginning, since no user is connected to the system, no session is created.

In the rest of this section, we consider three run-time changes of the system. Each change is represented by a snapshot that captures the system state. On each snapshot we check whether the constraints defined above are satisfied or violated.

First, let us consider the case in which user *Bob* gets a new job. Since he cannot participate to the mission anymore, he has to delegate, as a transfer, his role *securityOfficer* to *Alice*. To perform the delegation, he has to first activate one of his roles, e.g., role *manager*[5]. Figure 6 depicts, as an object diagram, an instance of the GemRBAC model that corresponds to a portion of the system state right after this delegation. Since the delegation of *Bob* to *Alice* is of type transfer, *Bob* is no longer member of role *securityOfficer*; the latter is assigned to *Alice* via a *role delegation* association. Notice that the object *officerDeleg* of type *Delegation*, having the attribute *isTransfer* set to *strong*, has been created. It keeps track of the delegated role (*securityOfficer*), the delegator user (*Bob*), the role of the delegator at the time of the delegation (*manager*), the delegated role (*securityOfficer*), the delegate user (*Alice*), and the role of the delegate (*participant*). Constraint **C4** states that right after a delegation of type strong transfer, the delegator is no longer assigned to the delegated role and all its juniors. This constraint can be expressed using the template (*StrongTransfer*) provided on page 23. As *Bob* is no longer a member of role *securityOfficer*, one can check that constraint **C4** is satisfied.

When users *Alice* and *Mallory* connect to the system, a new session is created for each of them, as shown in figure 7, with objects *sesAlice* and *sesMallory*. We assume that *Alice* is located in the mission geofence *AbayZone1*. The spatial constraint **C2** is satisfied and role *securityOfficer* is enabled. This constraint can be expressed using the template (*locationbased*) provided on page 25. Session *sesAlice* maps *Alice* to the roles that she is allowed to activate: *participant* and *securityOfficer*. The dependency

[5]Notice that the role being delegated is disabled because of constraint **C2**.

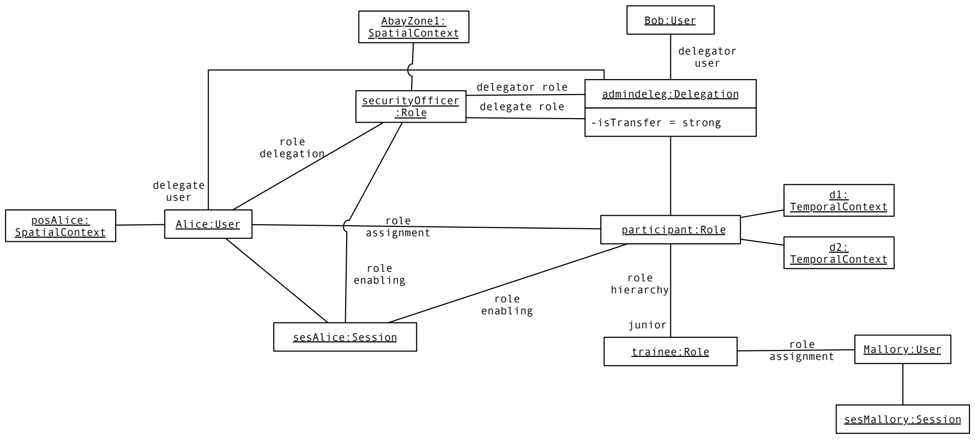


Figure 7: System state after the enabling of role *securityOfficer*

constraint **C3** states that role *trainee* is enabled if role *securityOfficer* is active. Since role *securityOfficer* is not active, *Mallory* cannot activate his role: as shown in figure 7, role *trainee* is not enabled in session *sesMallory*. This constraint can be expressed using the template (*RoleActivationDependency*) provided on page 24, by replacing the parameter r with role *trainee*, and role r_1 with role *securityOfficer*.

We now consider yet another change of the system corresponding to the instant when user *Alice* activates role *securityOfficer*. Consequently, constraint **C3** is satisfied and *Mallory* is allowed to activate role *trainee*. In this state, let us assume that *Alice* finds an injured person who needs medical treatment services. She sends an alert (permission *sendAlert*) to request help for the casualty. As shown in figure 8, a new object of the class *History* has been created to record the details of the operation: the user (*Alice*), the user's position (*posAlice*), the current time *t1*, the operation (*create*) performed on the object *alert*, the corresponding permission *sendAlert*, and the activated role *securityOfficer*.

Finally, the role *participant* will be disabled, according to constraint **C1**, when the mission ends (on day *d1*).

We remark that none of the existing RBAC models discussed in section 5 would be able to express the example presented above. More specifically, models RBAC96, RDM2000, PDM, Crampton et al., Sohr et al., TRBAC, GTRBAC, GEO-RBAC and LRBAC lack full support of context-based constraints. Consequently, constraints **C1** and **C2** could not be defined in these models. Although models LoTRBAC, STRBAC, and GSTRBAC cover time-based and location-based constraints, they do not support dependency and transfer policies. Consequently, constraints **C3** and **C4** could not be defined in these models.

The example above, based on a real world application, shows the lack of expressiveness of the models presented in the literature and illustrate the need for a unified modeling framework (like the proposed GemRBAC model), including constraint templates, to enable the specification and checking of a rich and realistic set of RBAC policies.

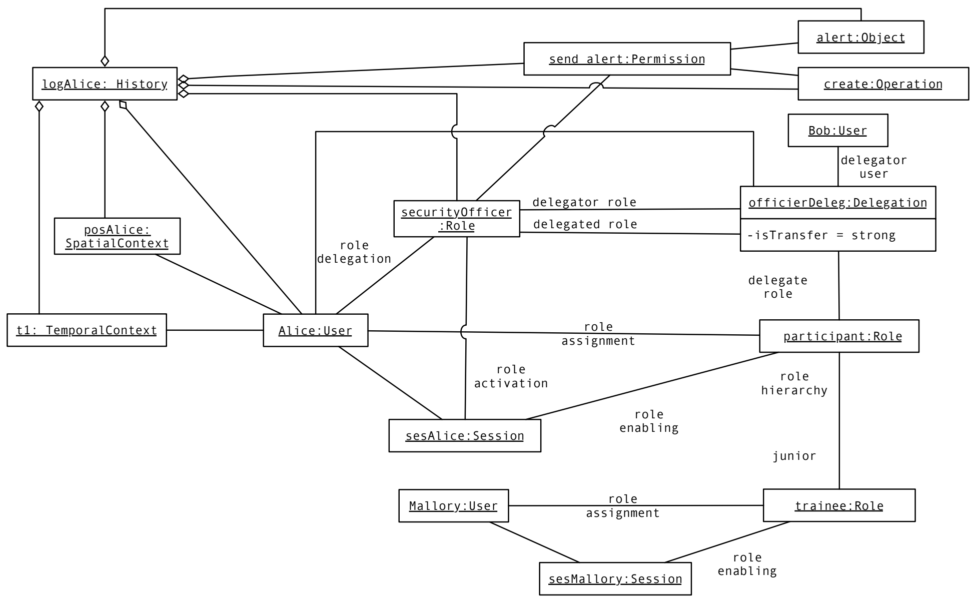


Figure 8: System state after sending an alert

9 Related Work

The topic of roles and their use in enterprise information systems has been discussed in a survey [46] that covers various aspects, ranging from roles in object modeling to roles in social psychology and management; the article also briefly summarizes some of the RBAC model extensions we have discussed in section 5. Reference [29] is a survey and classification of the scientific literature over 15 years of research on roles in information security. Reference [30] surveys the techniques used to verify RBAC policies, using both semi-formal approaches (mainly based on UML/OCL) and formal ones (based on specification languages like Z [41] and Alloy [16]). For a more general survey on access control services in operating systems, database management systems and network solutions (not limited to RBAC) we refer the reader to [13].

There have been several proposals for using OCL for the formalization of RBAC constraints [4, 43, 33, 23, 10, 22, 25]; however, the types of policies considered in each of these formalizations are a subset of the ones presented in this paper. In terms of model-based approaches for RBAC, SecureUML [28] is a modeling language for the model-driven development of secure systems, based on RBAC; it extends the original RBAC model to support authorization constraints, which are preconditions expressed in OCL, associated with operations that access system resources. In reference [6], authors present an RBAC model which combines SecureUML and ComponentUML. The latter is a UML-based language for modeling system entities and relationships between them. Authorization constraints are defined as OCL queries such as '*are there actions on concrete resources that every user can perform in the given scenario?*'. Similarly to our work, OCL queries are evaluated on the model instance which is a snapshot of the system state. However, we analyze the OCL constraints from a user's request point of view in order to make the access decision. The model-driven security approach proposed in [5] builds a security-aware graphical user interface model from the security model presented in [6] and a graphical user interface model. The goal is to automatically generate the graphical user interface application. Reference [24] shows how to incorporate RBAC policies into UML design models using UML diagram templates, but only supports role hierarchy and static and dynamic separation of duty constraints.

This paper fills the gap between the existing OCL-based formalizations of the RBAC policies and the various types of policies proposed in the literature, by formalizing *all* the types of policies classified in section 4 as OCL constraints on the GemRBAC model.

10 Conclusion and future work

RBAC is the de facto standard for access control in enterprise information systems: by assigning permissions to roles, it decouples users from permissions, simplifying the administration and deployment of access control in the enterprise. Since the original definition of RBAC in 1996, there have been many proposals to extend the original model to support various types of authorization constraints. However, there is no unified framework that can be used to define all these types of authorization constraints in a coherent way, using a common conceptual model. Moreover, imprecise definitions and semantics hinder the operationalization of some of these constraints.

In this paper we have proposed an extension of the original RBAC conceptual model, called GemRBAC, which includes all the entities required to express the various types of RBAC policies. These policies have been selected based on an analysis and classification of the various RBAC extensions proposed in the literature. The various RBAC policies have been formalized as OCL constraints on the UML representation of the GemRBAC model. This is expected to facilitate the selection and operationalization of policies, either at design or run time, based for example on OCL checkers. To support this, we make publicly available at `https://github.com/AmeniBF/GemRBAC-model` the Ecore version of the GemRBAC model, the OCL constraints defined in section 7, and the model instances that violate/satisfy them.

As part of future work, we plan to extend the GemRBAC with additional concepts, such as those related to administrative operations, and more advanced definitions of temporal and spatial contexts (e.g., periodicity expressions such as '*each Friday*' or relative location expressions such as '*5 miles around the current position of the user*'). For example, the latter could be realized by introducing a more detailed class hierarchy of class `RBACContext`. We also plan to define a domain-specific language to facilitate the definition of the various policies discussed in this paper. The policies defined with this language will then be mapped to the OCL constraints on the GemRBAC model; this mapping will enable the verification of the policies, both at design time and at run time, by means of OCL checkers, as hinted in section 8. Last, we plan to consolidate the GemRBAC model and the domain-specific language in a tool suite, to facilitate the concrete implementation of RBAC in organizations.

Acknowledgement

This work has been partially supported by the National Research Fund, Luxembourg (FNR/P10/03) and by a research grant from HITEC Luxembourg. Ameni Ben Fadhel is also supported by the Faculty of Science, Technology and Communication of the

University of Luxembourg. We thank Benjamin Hourte and Christophe Roussy from HITEC Luxembourg for their help in the definition of the example.