Int. J. Inf. Secur. (2010) 9:209–236
DOI 10.1007/s10207-010-0104-3

REGULAR CONTRIBUTION

A delegation model for extended RBAC

Meriam Ben-Ghorbel-Talbi · Frédéric Cuppens · Nora Cuppens-Boulahia · Adel Bouhoula

Published online: 1 May 2010

Abstract In the field of access control, delegation is an important aspect that is considered part of the administration mechanism. Thus, a comprehensive access control model must provide a flexible administration model to manage delegation and revocation. Unfortunately, to our best knowledge, there is no complete model for describing all delegation requirements for role-based access control. Therefore, proposed models are often extended to support new delegation or revocation characteristics, which is a complex task to manage and requires the redefinition of these models. Moreover, since delegation is modelled separately from administration, this requires the specification of a separate security policy to deal with delegation. In this paper, we describe a new delegation approach for extended role-based access control models. We show that our approach is flexible and is sufficient to deal with administration and delegation requirements in a homogeneous unified framework. Moreover, it provides means to express various delegation and revocation dimensions in a simple manner.

M. Ben-Ghorbel-Talbi (✉) · F. Cuppens · N. Cuppens-Boulahia
Institut TELECOM/TELECOM Bretagne, LUSSI, 2 Rue de la Châtaigneraie, CS 17607, 35576 Cesson Sévigné, Cedex, France
e-mail: meriam.benghorbel@telecom-bretagne.eu

F. Cuppens
e-mail: frederic.cuppens@telecom-bretagne.eu

N. Cuppens-Boulahia
e-mail: nora.cuppens@telecom-bretagne.eu

A. Bouhoula
Sup'Com Tunis, Digital Security Research Unit,
Route de Raoued Km 3.5, 2083 Ariana, Tunisia
e-mail: bouhoula@planet.tn

Keywords Administration · Delegation · Revocation · Role-Based Access Control · OrBAC model

1 Introduction

Delegation is the process whereby a user without any specific administrative prerogatives obtains the ability to grant some authorizations. In the field of access control, it is important to have a system that allows delegation in order to simplify the administrator task and to manage collaborative work securely, especially with the increase in shared information and distributed systems.

The access control system must be secure and flexible. Secure to guarantee that the security policy is not violated by delegation, and flexible to support delegation requirements, such as delegation of rights and obligations, temporary delegation, multiple and multi-step delegation, transfer, revocation. Moreover, the delegation process must be simple to manage, and we argue that it is important to deal with administration and delegation in a single framework.

In the literature, there are many works related to delegation in access control models. These works show that delegation is a complex problem to solve and is generally modelled separately from other administration requirements. The reason is that proposed models are generally based on the RBAC formalism [26] (Role-Based Access Control), which is not expressive enough to deal with all delegation requirements. Therefore, it is necessary to extend the RBAC model by adding new components, such as new types of roles, permissions and relationships. This is a complex task to manage, and to the best of our knowledge, there is no comprehensive model for describing all delegation requirements. Thus, delegation models themselves are extended to support new delegation characteristics such as [6] and [11], which are proposed to deal with the agent-based delegation and transfer, respectively.

In our earlier papers [8,9], we have showed that it is possible to express administration requirements, including delegation and revocation requirements, in a single model. In this paper, we aim at proposing a comprehensive framework to deal with delegation and revocation in role-based access control. Our work is based on an extended RBAC model that takes its inspiration in the OrBAC [1] (Organization-based Access Control) formalism. This model gives means to specify multi-granular and contextual authorizations [15], which facilitate the modelling of delegation and revocation characteristics such as temporary delegation, partial delegation, multiple and multi-step delegation, propagation, dominance, grant dependency.

Our model explicitly includes the concept of context which actually provides our model with high flexibility and expressiveness. Namely, by contrast to other models, we do not deal with each delegation and revocation level separately. Instead, using contextual permissions the administrator and also users may define complex conditions to deal with delegation and revocation features and to restrict the delegation scope.

At first, the administrator manages the right to delegate, transfer and revoke using "classic" privileges (i.e. it Permission and it Prohibition) and may specify constraints using contexts. These constraints may concern the grantor (i.e. the user who delegates the right) or the grantee (i.e. the user to which the right is delegated) characteristics, such as role, location, affiliation and previous actions, but also the delegated right (i.e. target, privilege) and the system attributes like time and circumstance. This kind of conditions is not supported by other models, since most of them only support constraints on the grantee's role. Some works are proposed to deal with new delegation and revocation constraints such as the grantee's attributes and the time. But these models lack flexibility, since each constraint level is defined separately.

Secondly, since the grantor delegates privileges (i.e. permissions and roles) just like the administrator assigns privileges to users, then he/she may also specify contextual delegation permissions. This feature provides high flexibility, since we can deal with several delegation features such as temporary, multiple and multi-step delegation only using contexts. Moreover, in our model, we consider that temporary delegation does not only concern time like in existing models, that associate a timeout to each delegation. We may also specify conditions on the grantee's location, actions, circumstances, etc. Hence, we extend the notion of temporary delegation to contextual delegation.

In addition, our model is based on an object-oriented approach, thus we do not manipulate privileges directly (i.e. it Permission and it Prohibition), but we use objects having a specific semantic. Each object corresponds to an assignment or a delegation of a privilege (e.g. a user to a role, a role or a user to a permission). This is very useful to manage the revocation of delegation, namely cascade and strong revocation (i.e. the revocation of the whole delegation chain and the revocation of all other delegations associated with the same grantee, respectively). In fact, this provides means to deal with the delegation chain in a simple manner, and there is no need to use a history relationship to record delegations.

This paper is organized as follows. In Sect. 2, we start with basic concepts of our extended RBAC model and its associated administration model. In Sect. 3, we present our delegation model. In Sect. 4, we give an overview of how revocation is managed in our model. We discuss in Sect. 5 the complexity and the decidability of our approach. Then, discussion and related work are given in Sect. 6. Finally, concluding remarks are made in Sect. 7.

2 Extended RBAC model

Our model is based on an extended RBAC model that aims to specify the security policy at the abstract level that is independently from the implementation of this policy. Thus, instead of modelling the policy by using the concrete concepts of subject, action and object, we suggest reasoning with the roles that subjects, but also actions and objects play in the organization. The role of a subject is simply called a role, whereas the role of an action is called activity, and the role of an object is called view. Moreover, this model explicitly includes the concept of context which is viewed as an extra condition that must be satisfied to activate a given security rule.

Our model takes its inspiration in the OrBAC formalism [1]. The central entity in OrBAC is the entity organization. Intuitively, an organization is any entity that is responsible for managing a security policy. But, in this paper, we assume that the policy applies to a single organization, and thus we omit the Org entity in the following.

Our model is compatible with a stratified Datalog with negation program [30]. Stratifying a Datalog program consists in ordering rules so that if a rule contains a negative literal then the rule that defines this literal is computed first. A stratified Datalog program is computable in polynomial time (see Sect. 5). To express rules and facts, we shall actually use a prolog-like notation. Terms starting with a capital letter, such as $Subject$, correspond to variables and terms starting with a lower case letter, such as $peter$, correspond to constants. A fact, such as:

$$Parent(peter, john).$$

says that $peter$ is a parent of $john$, and a rule such as:

$$Grand\_parent(X, Z) \text{:- } Parent(X, Y), Parent(Y, Z).$$

says that X is a grand-parent of Z if there is a subject Y such that X is a parent of Y and Y is a parent of Z.

Before presenting our delegation model, we briefly recall, in this section, the main components of our extended RBAC model.

2.1 Basic predicates

There are seven basic sets of entities: S (a set of subjects), A (a set of actions), O (a set of objects), R (a set of roles), $\mathscr{A}$ (a set of activities), V (a set of views) and C (a set of contexts). In the following, we present the basic built-in predicates:

- ***Empower*** is a predicate over domains SxR. If s is a subject and r is a role, then *Empower(s, r)* means that subject s is empowered in role r.
- ***Use*** is a predicate over domains OxV. If o is an object and v is a view, then *Use(o, v)* means that object o is used in view v.
- ***Consider*** is a predicate over domains Ax$\mathscr{A}$. If α is an action and a is an activity, then *Consider(α, a)* means that action α implements activity a.
- ***Hold*** is a predicate over domains SxAxOxC. If s is a subject, α is an action, o is an object and c is a context, then *Hold(s, α, o, c)* means that context c holds between subject s, action α and object o.
- ***Permission*** and ***Prohibition*** are predicates over domains R_sxA_axV_oxC, where $R_s = R \cup S$, $A_a = \mathscr{A} \cup A$ and $V_o = V \cup O$. More precisely, if g is a role or a subject, t is a view or an object and p is an activity or an action, then *Permission(g, p, t, c)* (resp. *Prohibition(g, p, t,c)*) means that grantee g is granted permission (resp. prohibition) to perform privilege p on target t in context c.
  These predicates enable to specify permissions and prohibitions at the abstract level, which contains roles, activities and views.
- ***Is_permitted*** and ***Is_prohibited*** are predicates over domains SxAxO. If s is a subject, α is an action and o is an object, then *Is_permitted(s, α, o)* (resp. *Is_prohibited(s, α, o)*) means that subject s is permitted (resp. is prohibited) to perform action α on object o.
  These predicates enable to specify permissions and prohibitions at the concrete level, which is based on subjects, actions and objects. A concrete permission or prohibition is respectively derived from an abstract permission or prohibition when the associated context holds.

2.2 Hierarchy and inheritance

In our model, it is suggested to define hierarchies [16] over roles but also activities and views and to associate permission inheritance with these different hierarchies. This is modelled as follows:

- Sub_role, is a partial order relation over domains RxR. If R_1 and R_2 are roles, then $Sub\_role(R_1, R_2)$ means that role R_1 is a sub-role (also called senior role) of role R_2. Permissions and prohibitions are inherited through the role hierarchy. For instance, inheritance of permissions is modelled by the following rule:
  Rule$_{Sub_R}$

  $Permission(R_1, A, V, C)$:-
  $Permission(R_2, A, V, C), Sub\_role(R_1, R_2)$.

- Sub_view, is a relation over domains V x V. If V_1 and V_2 are views, then $Sub\_view(V_1, V_2)$ means that V_1 is a sub-view of V_2.
  Rule$_{Sub_V}$

  $Permission(R, A, V_1, C)$:-
  $Permission(R, A, V_2, C), Sub\_view(V_1, V_2)$.

- $Sub\_activity$, is a relation over domains A x A. If A_1 and A_2 are activities, then $Sub\_activity(A_1, A_2)$ means that A_1 is a sub-activity of A_2.
  Rule$_{Sub_A}$

  $Permission(R, A_1, V, C)$:-
  $Permission(R, A_2, V, C), Sub\_activity(A_1, A_2)$.

- $Sub\_context$, is a relation over domains C x C. If C_1 and C_2 are contexts, then $Sub\_context(C_1, C_2)$ means that C_1 is a sub-context of C_2. This means that C_1 always holds between a subject, an action and an object when C_2 holds for the same subject, action and object.

How to manage inheritance of privileges through these different hierarchies is further explained in [16].

2.3 Global constraints

To specify global constraint in our model, we use the built-in predicate $Error$:

- $Error$ is a predicate over the null domain, i.e. a predicate of cardinality zero.

Using the $Error$ predicate, a global constraint corresponds to a logical rule whose conclusion is $Error$. If it is actually possible to derive $Error$ from some global constraints, then the security policy is not consistent.

To model separation of entity as global constraints, there are the following built-in predicates:

- $Separated\_role$ is a predicate over domains RxR. If R_1 and R_2 are roles, then $Separated\_role(R_1, R_2)$ means

that role R_1 is separated from role R_2. We assume that $Separated\_role$ is an irreflexive and symmetrical relation between roles.
Using this $Separated\_role$ predicate, we can then express the following global constraint:
Rule$_{Cst_R}$

$Error()$:-
  $Separated\_role(R_1, R_2)$,
  $Empower(S, R_1), Empower(S, R_2)$.

This constraint says that subject S cannot be empowered in both roles R_1 and R_2.
Like permissions and prohibitions, constraints are also inherited through the role hierarchy. This is modelled by the following rule:
Rule$_{SubCst_R}$

$Separated\_role(R_1, R_3)$:-
  $Separated\_role(R_1, R_2), Sub\_role(R_3, R_2)$.

– Similar predicates *Separated_view*(V_1, V_2), *Separated_activity* (A_1, A_2) and *Separated_context*(C_1, C_2) are introduced to specify the separation of views, activities and contexts, respectively.

2.4 Conflict management

Since in our model it is possible to specify both permissions and prohibitions, some conflicts may occur. We actually manage conflicts at the abstract level and provide sufficient conditions to guarantee that the absence of conflict at the abstract level also guarantees the absence of conflict at the concrete level.

Our approach [13] is based on defining conflict resolution strategy (CRS) that is used to assign priority levels to permissions or prohibitions. Then, predicates *Permission(g, p, t, c)* and *Prohibition(g, p, t, c)* are replaced by: *Permission(g, p, t, c, l)* and *Prohibition (g, p, t, c, l)*, where *l* is the priority level.

Two different situations may arise when using a CRS: (1) Redundant rules may exist and (2) potential conflict may arise. Redundant rules management is based on the notion of exception to a general authorization. This means that if rule R_i is a strict exception to rule R_j, then R_i should be assigned higher priority than rule R_j, else R_i never applies. To prevent redundant rules, we consider that $R_i(r_i, a_i, v_i, c_i)$ is a strict exception to $R_j(r_j, a_j, v_j, c_j)$ if the following condition holds:

$$sub\_role(R_i, R_j), sub\_activity(A_i, A_j),$$
$$sub\_view(V_i, V_j), sub\_context(C_i, C_j),$$
$$\neg(R_i = R_j, A_i = A_j, V_i = V_j, C_i = C_j).$$

Our approach to eliminate potential conflict consists in the detection of unrelated rules, by specifying separation constraints. For instance, if a separation constraint exists between roles r_1 and r_2, then a subject cannot be empowered into both roles r_1 and r_2. As a consequence, if a given permission is assigned to role r_1 and a given prohibition is assigned to role r_2, then these permission and prohibition cannot generate a conflict. Similarly, we can specify separation constraints between views, activities and contexts. We say that a permission and a prohibition are unrelated when such a separation constraint exists. It is then sufficient to assign priorities between every pair of permission and prohibition that are not unrelated to eliminate every potential conflict in the security policy. In the remainder of this paper, when there is no ambiguity, we omit the priority level attribute in the specification of permission or prohibition.

2.5 The administration model

Our model is self-administrated and unlike ARBAC [17], the administration model suggested for RBAC, we do not use specific relationships such *as can-assign* and *can-assignp*. That is the concepts used to define an administration policy are similar to the ones presented in the previous section.

Our administration model is based on an object-oriented approach, thus we do not manipulate privileges directly (i.e. $Permission$ and $Prohibition$), but we use objects having a specific semantic and belonging to specific views, called administrative views. Inserting an object in these views will enable to assign permissions, prohibitions or roles to users.

Before presenting our administration model, we give in the following the basic concepts of object and view.

The Notion of Object The administration model follows an object-oriented approach. Thus, every entity corresponds to an object (see Fig. 1). Each object has an identifier that uniquely identifies the object and a set of attributes to describe the object.

Following a logical formalism, attributes are modelled as binary predicates. For instance, fact *Age*(s_1*, 20).* says that the object whose identifier is s_1 has an attribute age whose value is 20. Objects belong to classes. This is modelled using a binary predicate *Class*. For instance, the fact *Class*(s_1*, student).* says that $s_1$belongs to the student class.

The Notion of View The concept of View is different from the concept of Class. A class is a taxonomical concept used to structure the object description, whereas a view is an organizational concept used to structure the policy specification.

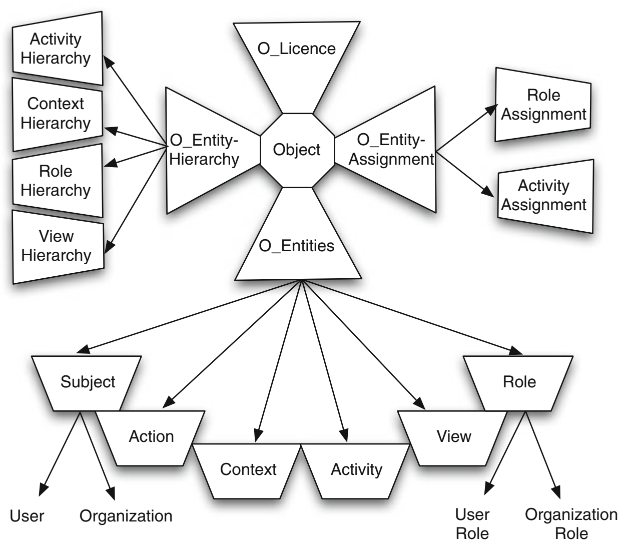


Fig. 1 Administration model

In our model, objets are used into views. This is modelled by a binary predicate *Use*. For instance,

$Use(c_1, course)$.

says that object c_1 is used in view *course*.

A view may be defined by enumerating facts specifying which objects are used in this view or by a logical rule called a *view definition*. For instance:

$Use(C, graduate\_course)$:-
$\quad Use(C, course), level(C, graduate)$.

This rule says that object C is used in view *graduate_course* if C is used in view *course* and the attribute *level* of C is equal to graduate.

Thus, a view definition corresponds to a condition which is used to automatically derive that some objects belong to a given view.

When a subject creates an object *obj*, then a view v must be specified. If the security policy permits that this subject creates this object in this view, then the following fact is inserted: $Use(obj, v)$.

Managing administrative privileges

The approach in the administration model [15] is to define administration functions by considering different administrative views. Objects belonging to these views have specific semantics; More precisely, we shall consider in the following two administrative views called *role_assignment* and *licence* views. They are respectively interpreted as an assignment of a user to a role in the *role_assignment* view and a permission (or a prohibition) to a role or to a user in the *licence* view.

Intuitively, inserting an object in these views will enable an authorized user to assign a user to a role, assign a permission to a role or assign a permission to a user, respectively. Conversely, deleting an object from these views will enable a user to perform a revocation. Defining the administration functions then correspond to specifying which role is permitted to have an access to the administrative views. So that only valid licences can be created.

Licence view This view is used to specify and manage the security policy. Objects belonging to the *licence* view have the following attributes: *Grantee*: subject to which the licence is granted, *Privilege*: action permitted by the licence, *Target*: object to which the licence grants an access and *Context*: specific conditions that must be satisfied to use the licence.

The existence of a valid licence is interpreted as a permission by the following rule:

Rule$_{LA}$

$Permission(Sub, Act, Obj, Context)$:-
$\quad Use(L, licence), Grantee(L, Sub),$
$\quad Privilege(L, Act), Target(L, Obj),$
$\quad Context(L, Context)$.

In addition, we define the following **Rule**$_{LA'}$ for the sub-views of view *licence*. This means that objects inserted in the sub-view of *licence* are also interpreted as permissions.

Rule$_{LA'}$

$Use(L, licence)$:-
$\quad Use(L, Sub\_licence), Sub\_view(Sub\_licence, licence)$.

Contextual licence As we have mentioned, the licence class is associated with an attribute called *context*. Contexts are used to specify conditions, for example working hours, during vacation or urgency.

Conditions that must be satisfied to derive that a context is active are modelled by a logical rule called context definition. For instance, let us consider the following rule:

$Hold(Sub, Act, Obj, during\_vacation)$:-
$\quad In\_vacation(Sub)$.

This context definition simply says that a subject executes an action on an object in context *during_vacation* if this subject is in vacation.

In [12], five kinds of contexts have been defined:

- the Temporal context that depends on the time at which the subject is requesting for an access to the system,
- the Spatial context that depends on the subject location,
- the User-declared context that depends on the subject objective (or purpose),

– the Prerequisite context that depends on characteristics that join the subject, the action and the object.
– the Provisional context that depends on previous actions the subject has performed in the system.

We can also combine these elementary contexts to define new composed contexts by using conjunction, disjunction and negation operators: &, $\oplus$ and $\bar{}$. This means that if c_1 and c_2 are two contexts, then $c_1 \& c_2$ is a conjunctive context, $c_1 \oplus c_2$ is a disjunctive context and $\bar{c_1}$ is a negative context.

In the following, we shall also consider a context called *nominal* that is always active for any subject, action and object.

We need another rule to derive actual permission for some subject, action and object when the context is active. This is defined by the following logical rule:

Rule$_C$

$$\begin{aligned}&Is\_permitted(Sub, Act, Obj)\text{:-}\\&\quad Permission(Sub, Act, Obj, Context),\\&\quad Hold(Sub, Act, Obj, Context).\end{aligned}$$

Role_assignment view This view is defined to manage the assignment of subjects to roles. Objects belonging to this view are associated with the following attributes: *Assignee*: subject to which the role is assigned and *Assignment*: role assigned by the role assignment. Following is the rule to interpret objects of view *role_assignment*:

Rule$_{RA}$

$$\begin{aligned}&Empower(Subject, Role)\text{:-}\\&\quad Use(RA, role\_assignment), Assignee(RA, Subject),\\&\quad Assignment(RA, Role).\end{aligned}$$

Similarly to view *licence*, we define the following rule to interpret objects of the sub-views of *role_assignment*.

Rule$_{RA'}$

$$\begin{aligned}&Use(RA, role\_assignment)\text{:-}\\&\quad Use(RA, Sub\_role\_assignment),\\&\quad Sub\_view(Sub\_role\_assignment, role\_assignment).\end{aligned}$$

Multi-granular licence The attribute *grantee* of a licence can be a subject or a role, the attribute *privilege* can be an action or an activity, and the attribute *target* can be an object or a view. If the grantee is a role, we need an additional rule to derive that a given permission is granted to a subject, when a permission is granted to a role and this subject is assigned to this role:

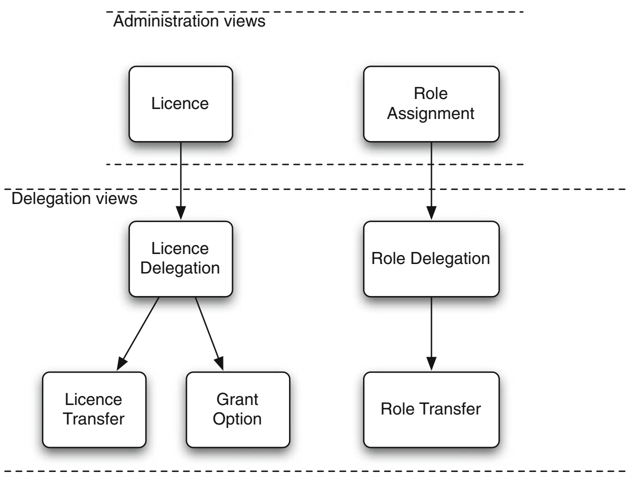


Fig. 2 Delegation views

Rule$_{Mg}$

$$\begin{aligned}&Permission(Subj, Act, Obj, Context)\text{:-}\\&\quad Permission(R, Act, Obj, Context),\\&\quad Use(R, role), Empower(Sub, R).\end{aligned}$$

There are two other similar rules to interpret licences when the attribute privilege is an activity and the attribute target is a view, respectively.

3 The delegation model

3.1 Introduction

We present in this section our delegation model. Our main contribution is to propose a self-administrated model that is flexible enough to manage different delegation characteristics and simple to use and to administrate.

Our model is an extension of the administration model defined in Sect. 2.5. Note that we do not define new components, we have simply to consider new administrative views, which we call *delegation views* (see Fig. 2). To manage the delegation of rights (permissions or roles), we define sub-views of *licence* and *role_assignment* views called *licence_delegation* and *role_delegation*, respectively. To manage the transfer of rights (i.e. to delegate permissions or roles and the grantor loses the delegated permissions or roles during the delegation), we define sub-views of *licence_delegation* and *role_delegation* called *licence_transfer* and *role_transfer*, respectively. Finally, to manage the delegation of the delegation rights (i.e. to delegate the right to delegate permissions or roles), we define a sub-view of *licence_delegation* called *grant_option*. We discuss hereafter some assumptions before presenting the structure of these delegation views.

We consider that delegation can be classified as the delegation of right or the delegation of obligation. Right delegation allows a user, called the grantor, to delegate his/her permissions (partial delegation) or roles (total delegation) to another user called the grantee. So the grantee, which is either a subject or a role, is allowed to perform the delegated permissions or roles on behalf of the grantor.

The delegation of obligation allows a grantor to delegate his/her obligations to a grantee with his/her agreement, in the case of bilateral agreement delegation, or without [10]. So, the grantee must perform the delegated obligations on behalf of the grantor. In the remainder of this paper, we only focus on the delegation of right, further works will be dedicated to the delegation of obligations.

The grantor or some specific authorized users must have the possibility to revoke the delegation. This revocation can be performed manually or automatically, with simple or cascade propagation and with strong or weak dominance. In this paper, we give an overview on how we manage the right to revoke the delegation and how we deal with revocation schemes such as propagation and dominance.

When revocation is handled automatically, the delegation is called temporary. In this case, the grantor specifies the context in which this delegation applies.

The delegation can also be redelegated: the grantee can be allowed to redelegate a given delegation to another user (in the case of multi-step delegation). In this situation, the grantor must have the possibility to control the chain of delegation by specifying the depth of this chain.

The delegation action can be performed either by the grantor (self-acted) or by a third party (agent acted), this means that the grantor delegates the administration of his/her right to another user. So, this user can delegate the right of the grantor (a permission or a role) on his/her behalf. We consider this situation as a particular case of multi-step delegation when the delegation depth is equal to 1.

Upon delegation, the grantor can maintain the permission or the role he/she has delegated, or loses it for the duration of the delegation. This kind of delegation is called non-monotonic or also transfer.

We consider that it is important to give the possibility to the administrator to manage the delegation policy with a simple manner. Since, one of the main delegation objectives is to simplify the administrator task.

Managing the delegation policy consists in specifying which principal (role or subject) is permitted to delegate his/her rights (permissions or roles) and in which context. The context is an important aspect to consider, because it allows the definition of many different conditions useful in delegation, such as temporal or prerequisite conditions.

3.2 Motivating example

We give in this section a motivating example to be used throughout the paper to illustrate the various aspects of our approach.

Let us consider an organization such as *TELECOM Bretagne* with the following entities and security rules:

Roles and subjects

– Director: *Bob*,
– Head of department: *Alex*,
– Professor: *Paul* and *Mary*,
– Assistant: *Bill*.

Views

– Stud_mark: contains student's marks files,
– Courses_files: contains courses files,
– Timetable: contains the timetable database.

Security rules

$Permission(professor, manage, stud\_mark, own\_students).$
$Permission(professor, manage, courses\_files, own\_courses).$
$Permission(head\_dep, manage, timetable, nominal).$

where contexts *own_students* and *own_courses* are provisional contexts meaning that professors are only allowed to manage the marks and the courses of students that they teach, respectively.

In this organization, users may need to delegate some of their authorities to other users. For instance, a professor may need the help of an assistant to achieve a given work on his/her behalf or to perform a collaborative work. The head of department may need the help of a professor to replace him/her during his/her absence. We consider in the following some delegation situations:

1. A professor can delegate or transfer the proof-reading of his/her student's examinations to his/her assistant.
2. A professor can delegate or transfer his/her teaching task to an assistant.
3. The head of department can delegate or transfer his/her role to another professor during his/her absence.
4. The director can delegate on behalf of the head of department his/her role (or some *head_dep*'s permissions) to a professor (e.g. when the head of department is absent).
5. If a professor empowered in role *head_dep* by delegation is no longer able to perform this task, then he/she can delegate, in his/her turn, this role to another professor.

We also consider some delegation constraints to control these delegation situations:

6. A professor is allowed to delegate the proof-reading of his/her student's examinations only to (a) his/her assistant and (b) to one assistant at a given time.
7. The head of department is allowed to delegate his/her role only (a) to a professor and (b) to one professor at a given time.
8. A professor delegates the proof-reading of his/her student's examinations only during the examination period.
9. A professor allows an assistant to proof-read his/her student's examinations only in his/her office.
10. *Alex* may delegate his role and permissions only to *Mary*.
    Similarly, users may revoke their delegated permissions or roles according to their needs, and the administrator must be able to specify revocation constraints to control the right to revoke. We give hereafter some revocation constraints:
11. Users are allowed to revoke their own delegations.
12. The head of department can revoke professors empowered in his/her role by delegation, even if he/she is not the grantor of this delegation (e.g the grantor is the director or another professor).
13. The director is allowed to revoke users from their delegated roles.
14. The director is allowed to revoke role transfer only if the grantor of this transfer is not in vacation.

These different situations and conditions will be used in the following sections to further explain our approach and to show how we are dealing with delegation requirements in our model.

3.3 Delegation views

In this section, we present the structure of the delegation views and further analyze the administration functions associated with the management of these views.

3.3.1 The view licence_delegation

We define a delegation view called *licence_delegation* to manage the delegation of permissions. As shown in Fig. 2, this is a sub-view of view *licence*, thus objects belonging to it inherit the attributes and the semantic of *licence* objects.

These objects have also an additional attribute called *Grantor*: subject who is delegating the licence. This attribute is important to revoke the delegation and to recover the grantor privileges when the transfer is revoked.

As we have mentioned earlier, the grantee can be a subject or a role, the privilege can be an action or an activity, and the target can be an object or a view. This is an important aspect of our model which gives means to specify multi-granular privileges. So, the grantor can delegate his/her permission to a grantee which is either a subject or a role. This means that our model supports User-to-User (U2U) and User-to-Role (U2R) delegation.

According to $\mathbf{Rule}_{LA'}$, inserting an object in the *licence_ delegation* view will enable an authorized grantor to delegate a permission to a grantee. This means that this view allows partial delegation.

Example 1 **Collaboration of work** Let us consider the first delegation situation of our example, when professor *Paul* needs the help of his assistant *Bill* to proof-read his student's examination. The role assistant is generally not permitted to have an access to the student's marks. Then, *Paul* should delegate to *Bill* a permission to access his students marks. For this purpose, *Paul* should create the following object:

```
Grantor: Paul,
Grantee: Bill,
Target: Paul_stud_mark,
Privilege: Access,
Context: Nominal.
```

Then, he should insert this object in the *licence_ delegation* view. According to $\mathbf{Rule}_{LA'}$, the following permission is created:

$$Permission(bill, access, paul\_stud\_mark, nominal).$$

We assume that *paul_stud_mark* is a sub-view of *stud_ mark* containing *Paul*'s student files.

Obviously, *Paul* must have a permission to delegate to *Bill* the right to access his student's marks. This kind of permissions is specified by the administrator in the delegation policy. We present in the following how to manage the partial delegation.

Managing partial delegation To manage the delegation policy, the administrator must define which grantor *Gr* (role or subject) has an access to the *licence_delegation* view and in which context *C*. This is defined by facts having the following form:

$$Permission(Gr, delegate, licence\_delegation, C). \quad (1)$$

$$Permission(Gr, delegate, sub\_licence\_delegation, C). \quad (2)$$

where *sub_licence_delegation* is a sub-view of *licence_ delegation* defined as follows:

$$\begin{aligned}&Use(L, sub\_licence\_delegation)\text{:-}\\&\quad Use(L, licence\_delegation),\\&\quad Licence\_Condition.\end{aligned}$$

Licence_Condition corresponds to a set of conditions on the licence attributes, i.e. the grantor, the grantee, the target, the privilege and/or the context.

Fact (1) means that a grantor *Gr* is allowed to insert an object in the view *licence_delegation* when context *C* holds.

Fact (2) means that a grantor *Gr* is allowed to insert an object in the view *sub_licence_delegation* when context *C* holds and only if this object corresponds to the definition of this view: the inserted object *L* must satisfy *Licence_ Condition* (an example will be given hereafter).

Note that thanks to the use of views the administrator can express a large number of conditions useful in delegation. These conditions consist in prerequisite conditions regarding the grantor, the grantee, the target, the privilege or/and the context. This provides higher expressivity than other delegation models, where the prerequisite conditions only concern the role of the grantee (such as in RBDM [4], PBDM [33], RDM2000 [32]), or his/her attributes (such as in ABDM [31]).

Besides these conditions, the administrator can express extra conditions thanks to the use of contexts. Indeed, OrBAC provides different types of contexts such as temporal, spatial, prerequisite and provisional. More details will be given in Sect. 3.4.

Example 2 To illustrate the management of the *licence_delegation* view, let us consider a situation where the administrator wants to allow all users to delegate their permissions. This is defined using the permission:

$$Permission(default\_Role, delegate, licence\_delegation,\\ valid\_licence\_grantor).$$

where *default_Role* is a role in which all authorized users are empowered and context *valid_licence_grantor* is defined as follows:

$$\begin{aligned}&Hold(U, delegate, L, valid\_licence\_grantor)\text{:-}\\&\quad Use(L, licence\_delegation), Target(L, O),\\&\quad Privilege(L, A), Context(L, C),\\&\quad Permission(U, A, O, C).\end{aligned}$$

This context means that any user is allowed to delegate a permission if this user has this permission.

The administrator can also express prerequisite conditions using views. For instance, in *Example 1*, we have said that *Paul* must have a permission to delegate a right to *Bill*. For this purpose, the administrator should give a permission to role *Professor* to delegate an access to his student's marks. He may specify a prerequisite condition like the following rule:

$$Permission(professor, delegate, stud\_mark\_deleg,\\ valid\_licence\_grantor).$$

where sub-view *stud_mark_deleg* is defined as follows:

$$\begin{aligned}&Use(L, stud\_mark\_deleg)\text{:-}\\&\quad Use(L, licence\_delegation), Grantee(L, GR),\\&\quad Target(L, V), Empower(GR, teacher),\\&\quad Use(V, stud\_mark).\end{aligned}$$

This means that role *Professor* is allowed to insert an object in view *stud_mark_deleg*, and this object must satisfy the definition of this view: the grantee must be a teacher and the target must be a student's marks file. This permission is activated only if the context *valid_licence_grantor* holds, thus a professor is only allowed to delegate access to his/her student's marks.

As discussed in this section, view *licence_delegation* and its sub-views are dedicated to the partial delegation (i.e. permission delegation). In the following Sect. 3.3.2, we present another view to deal with total delegation.

3.3.2 The view role_delegation

This is a sub-view of *role_assignment*, thus objects belonging to this view inherits the semantic and the attributes of view *role_assignment*. But also have an additional attribute called *Grantor*: the subject who is delegating the role.

According to $\boldsymbol{Rule}_{RA'}$ inserting an object in this view will enable an authorized grantor to delegate a role to a grantee.

Example 3 **Backup of role** Let us consider the third situation of our example and assume that *Alex* the head of department will be absent for several days, and he wants to delegate his role during his absence to professor *Mary*. For this purpose, he should create the following object:

Grantor: Alex, Assignee: Mary, Assignment: Head_dep.

Then, he should insert this object in the *role_delegation* view. According to $\boldsymbol{Rule}_{RA'}$, the following fact is created:

$$Empower(mary, head\_dep).$$

Managing total delegation

Similar to the partial delegation, to manage total delegation the administrator must define which grantor *Gr* (role or subject) has an access, in this case, to the *role_delegation* view and in which conditions. This is defined by facts having the following forms:

$$Permission(Gr, delegate, role\_delegation, C). \quad (1)$$

$$Permission(Gr, delegate, sub\_role\_delegation, C). \quad (2)$$

where *sub_role_delegation* is a sub-view of *role_delegation* defined as follows:

$$Use(RD, sub\_role\_delegation)\text{:-}$$
$$Use(RD, role\_delegation), Role\_Condition.$$

Role_Condition corresponds to a set of conditions on the object attributes, i.e. the grantor, the assignee and/or the assignment.

Fact (1) means that a grantor *Gr* is allowed to insert an object in the *role_delegation* view when context *C* holds.

Fact (2) means that a grantor *Gr* is allowed to insert an object in the *sub_role_delegation* view when context *C* holds and only if this object satisfies *Role_Condition*.

Example 4 To explain the management of the *role_delegation* view, let us consider that the administrator wants to allow all users to delegate their roles. This can be defined using the prerequisite context *valid_role_grantor* as follows:

$$Permission(default\_Role, delegate, role\_delegation, valid\_role\_grantor).$$

where context *valid_role_grantor* is defined as follows:

$$Hold(U, delegate, RD, valid\_role\_grantor)\text{:-}$$
$$Use(RD, role\_delegation), Grantor(RD, U),$$
$$Assignment(RD, R), Empower(U, R).$$

This context means that any user empowered in a role *R* is allowed to delegate *R*.

As explained in the management of the partial delegation, the administrator can also specify conditions thanks to the use of sub-views. For instance, to specify the delegation constraint ***7-a*** described in our example, the administrator must give the role *Head of department* the permission to delegate his/her role as follows:

$$Permission(head\_dep, delegate, head\_dep\_deleg, nominal).$$

$$Use(RD, head\_dep\_deleg)\text{:-}$$
$$Use(RD, role\_delegation), Assignee(RD, Gr),$$
$$Assignment(RD, head\_dep),$$
$$Empower(Gr, professor).$$

This means that role *Head of department* is allowed to delegate his/her role only to a professor.

The two views described up to now are used in the monotonic delegation case, this means that upon delegation the grantor maintains the permission or the role he/she has delegated. In the following, we define two delegation views to deal with non-monotonic delegation, i.e. the transfer of rights (permissions and roles).

3.3.3 The view licence_transfer

We define the delegation view called *licence_transfer* to transfer a permission to a user. This view is a sub-view of the *licence_delegation* view. Thus, inserting an object in *licence_transfer* will create a new permission to the grantee according to ***Rule***$_{LA'}$, similar to view *licence_delegation*.

In addition, the grantor will lose the permission he/she has delegated during the delegation. In our approach, this permission is not physically removed. Instead, the transfer will activate a prohibition to the grantor, according to ***Rule***$_{LTr}$. As suggested is Sect. 2.4, a priority level is used to solve conflict that occurs between the activated prohibition and the other permissions. Since this prohibition must always override the other conflicting permissions, this prohibition is associated with the highest priority level which we call *high_priority*.

Rule$_{LTr}$

$$Prohibition(Sub, Act, Obj, C, high\_priority)\text{:-}$$
$$Use(L, licence\_transfer), Grantor(L, Sub),$$
$$Privilege(L, Act), Target(L, Obj), Context(L, C).$$

Note that the context of the prohibition and the delegated permission is the same one. So the grantor will lose this permission only for the time of the delegation.

Example 5 **Permission transfer.** We consider the second situation of our example, and we assume that professor *Paul* wants to grant the teaching task of the computer science course to his assistant *Bill*. In this case, *Paul* is no longer responsible of this course, so he should transfer to *Bill* the access to this course files. For this purpose, *Paul* must create and insert the following object in the *licence_transfer* view:

Grantor: Paul,
Grantee: Bill,
Privilege: Access,
Target: Comp_files,
Context: Nominal.

where *comp_files* is a sub-view of *courses_files* containing the computer science course files.

According to ***Rule***$_{LA'}$, the following permission is created for *Bill*:

$$Permission(bill, access, comp\_files, nominal).$$

And according to ***Rule***$_{LTr}$, the following prohibition is created for *Paul*:

$$Prohibition(paul, access, comp\_files, nominal, high\_priority).$$

Managing partial transfer To manage non-monotonic delegation (or transfer), the administrator must specify which users have an access to view *licence_transfer*. This is defined as follows:

$Permission(Gr, transfer, licence\_transfer, C).$
$Permission(Gr, transfer, sub\_licence\_transfer, C).$

where *sub_licence_transfer* is a sub-view of *licence_transfer* defined as follows:

$Use(L, sub\_licence\_transfer)$:-
    $Use(L, licence\_transfer), Licence\_Condition.$

Managing this view is completely similar to the management of the *licence_delegation* view, so we do not give more details here. But we give hereafter another example of prerequisite context to illustrate how much the notion of context is useful to deal with delegation.

Example 6 Let us assume that the administrator wants to allow the role *Professor* to transfer his/her rights but only to his/her assistant (delegation constraint ***6-a***). This is defined by the following facts:

$$Permission(professor, transfer, licence\_transfer, valid\_licence\_grantor \,\&\, assistant).$$

where the context *assistant* is defined as follows:

$Hold(U, transfer, L, assistant)$:-
    $Use(L, licence\_transfer), Grantor(L, U),$
    $Grantee(L, U'), Is\_assistant(U', U).$

We assume that $Is\_assistant(S_1, S_2)$ is an application-dependent predicate meaning that S_1 is an assistant of S_2.

3.3.4 The view role_transfer

This view is used to manage role transfer. It is a sub-view of the *role_delegation* view, so inserting an object in it will create an assignment of a grantee to a role, according to ***Rule***$_{RA'}$. To revoke the grantor from the transferred role, we have to modify the *role_Assignment* rule (***Rule***$_{RA}$) as follows:

Rule$_{RA}$

$Empower(Gr, R)$:-
    $Use(RA, role\_assignment), Assignee(RA, Gr),$
    $Assignment(RA, R),$
    $\neg(Use(RD, role\_transfer), Grantor(RD, Gr),$
    $Assignment(RD, R)).$

This rule means that a user is empowered in a role only if this user does not transferred his/her role. Thus, when the grantor Gr transfers his/her role R then, he/she will no longer be empowered in R. This user will be empowered again in role R when object RD is removed from the *role_transfer* view, i.e. when the role transfer ends.

Example 7 **Role transfer** Let us turn back to *Example 3* and suppose that the head of department *Alex* will take a sabbatical year. So, he must transfer his role to another user since, during this year, he is no longer responsible for this role. For this purpose, he must insert the same object, created in *Example 3*, in view *role_transfer*, in this case. This will create the same fact as previously:

$$Empower(mary, head\_dep)$$

But, according to ***Rule***$_{RA}$, $Alex$ is no longer empowered in role $Head\_dep$, because it is transferred to $Mary$.

Managing role transfer In the same way as previous views, to manage role transfer the administrator must specify users having access to view *role_transfer*. This is defined as follows:

$Permission(Gr, transfer, role\_transfer, C).$
$Permission(Gr, transfer, sub\_role\_transfer, C).$

where *sub_role_transfer* is a sub-view of *role_transfer* defined as follows:

$Use(RD, sub\_role\_transfer)$:-
    $Use(RD, role\_transfer), Role\_Condition.$

Since managing this view is similar to the management of view *role_delegation*, we do not give here more explicative details.

3.3.5 The view grant_option

We deal in this section with the delegation of delegation rights. This means that the grantor allows the grantee to redelegate a given delegation (a licence or a role) to another user. The redelegation is controlled by specifying the depth of the delegation chain. If the depth is equal to 1 the grantee is only allowed to redelegate the right, and if it is greater the grantee is also allowed to give another user the permission to redelegate the right. The *grant_option* view is defined to deal with this delegation aspect.

Like *licence_transfer*, *grant_option* is a sub-view of the *licence_delegation* view. Thus, objects belonging to this view have the same attributes and semantic as the *licence_delegation* view, but also have an additional attribute called *Step*: the depth of the delegation chain. Note that the privilege and the target of these objects concern delegation activities and delegation views, respectively.

According to $\textbf{\textit{Rule}}_{LA'}$, inserting an object in the *grant_option* view allows the grantee to delegate or transfer a licence or a role depending on whether the target is a sub-view of *licence_delegation*, *role*_delegation, *licence_transfer* or *role_transfer*.

The attribute *step* is used to control the propagation of the right to delegate. When the step is greater than 1, the grantee is allowed to propagate the delegation right (i.e. the right to delegate a licence or a role) to another user. This is defined by the following rule:

$\textbf{Rule}_{Gop}$

$$\begin{aligned}
&Permission(GR, delegate, grant\_option, C\& \\
&\qquad\qquad valid\_redelegation(LG)):\text{-} \\
&\quad Use(LG, grant\_option), Grantee(LG, GR), \\
&\quad Context(LG, C), Step(LG, N), N > 1.
\end{aligned}$$

The context *valid_redelegation* is defined as follows:

$$\begin{aligned}
&Hold(U, delegate, LG, valid\_redelegation(LG')):\text{-} \\
&\quad Use(LG, grant\_option), \\
&\quad Sub\_Licence(LG, LG'), \\
&\quad Step(LG, N), Step(LG', N'), \\
&\quad N < N'.
\end{aligned}$$

The prerequisite context *valid_redelegation* is very important since it controls the delegation chain. The grantee is allowed to delegate only the right or a sub-right of the right he/she was received and with a lower step. So, we can be sure that the redelegated right will never be a super-right that grants more privileges and/or with a longer delegation chains than the original right delegated by the first grantor.

For instance, if we assume that user *A* delegates a right *R* to user *B* with a step equals to *N*, and *B* redelegates this right to user *C* with a step equals to *N*′, then we are sure that the right received by *C* is at most comparable to right *R* and step *N*′ is lower than *N*. The delegation chain is stopped when the step is equal to 0.

The predicate *Sub_Licence* is defined as follows:

$$\begin{aligned}
&Sub\_Licence(L, L'):\text{-} \\
&\quad Use(L, licence), Use(L', licence), \\
&\quad Target(L, T), Target(L', T'), \\
&\quad Sub\_Target(T, T'), \\
&\quad Privilege(L, P), Privilege(L', P'), \\
&\quad Sub\_Privilege(P, P'), \\
&\quad Context(L, C), Context(L', C'), \\
&\quad Sub\_Context(C, C').
\end{aligned}$$

We consider *O* is a *Sub_Target* of *O'* if they are two views and *O* is a *Sub_View* of *O'*, or if *O* is an object used in the view *O'*, or if they are equal. Notice that the operator ';' corresponds to a disjunction.

$$\begin{aligned}
&Sub\_Target(O, O'):\text{-} \\
&\quad Sub\_View(O, O'); Use(O, O'); O = O'.
\end{aligned}$$

Similarly, we define the predicate *Sub_Privilege*. Predicate *Sub_context* was already defined in Sect. 2.2.

Example 8 **Agent-acted delegation** We consider situation ***4*** of our example, and we assume that the head of department *Alex* wants to give to director *Bob* the permission to delegate his role on his behalf when he is absent. But he specifies that this role must be delegated only to professor *Mary* (according to delegation constraint ***10***). For this purpose, he must create and insert the following licence *alex_to_bob* in the view *grant_option*:

Grantor: Alex,
Grantee: Bob,
Privilege: Delegate,
Target: Rd_view,
Context: Absent(Alex),
Step: 1.

where target *rd_view* is a sub-view of *role_delegation* defined as follows:

$$\begin{aligned}
&Use(RD, rd\_view):\text{-} \\
&\quad Use(RD, role\_delegation), Assignee(RD, mary), \\
&\quad Assignment(RD, head\_dep).
\end{aligned}$$

The following permission is created according to $\textbf{\textit{Rule}}_{LA'}$:

$$Permission(bob, delegate, rd\_view, absent(alex)).$$

We can also consider the case where *Alex* allows *Bob* to delegate an individual permission to *Mary* during his absence, for instance the access to timetable database. For this purpose, he must create the same object as previously with another target *ld_view* defined as follows:

$$\begin{aligned}
&Use(L, ld\_view):\text{-} \\
&\quad Use(L, licence\_delegation), Grantee(L, mary), \\
&\quad Privilege(L, access), Target(L, timetable).
\end{aligned}$$

In the same way, the following permission is created:

$$Permission(bob, delegate, ld\_view, absent(alex)).$$

Note that *Alex* can also allow *Bob* to transfer his permissions or roles. For this purpose, he must proceed as previously described but he must specify privilege “transfer” instead of “delegate”.

Example 9 **Multi-step delegation** Let us assume that *Alex* gives to *Bob* the permission to delegate his role to another user and also to allow this user to redelegate this role (according to situation **5** of our example). *Alex* specifies the following conditions: the redelegation applies only during his vacation, the user must be a professor, and the depth of the delegation chain is limited to 3 steps. For this purpose, he must create the following licence $alex\_to\_bob_2$:

Grantor: Alex,
Grantee: Bob,
Privilege: Delegate,
Target: Head_view,
Context: Vacation(Alex),
Step: 3.

where target *head_view* is defined as follows:

$$
\begin{aligned}
&Use(RD, head\_view)\text{:-}\\
&\quad Use(RD, role\_delegation),\\
&\quad Assignee(RD, GR),\\
&\quad Empower(GR, professor),\\
&\quad Assignment(RD, head\_dep).
\end{aligned}
$$

Inserting this object in the view *grant_option* will create the following permissions (the first permission is derived from $\boldsymbol{Rule}_{LA'}$, the second one is derived from $\boldsymbol{Rule}_{Gop}$):

$$
\begin{aligned}
&Permission(bob, delegate, head\_view, vacation(alex)).\\
&Permission(bob, delegate, grant\_option, vacation(alex)\\
&\qquad \& valid\_redelegation(alex\_to\_bob_2)).
\end{aligned}
$$

So, *Bob* is allowed to delegate role *Head of department* to any professor during *Alex* vacation. And he can also allow the assignee to delegate role *Head of department* when *Alex* is absent.

To illustrate this, we assume that *Alex* is absent, so *Bob* wants to delegate to professor *Paul* the role *Head of department*, and he wants to allow *Paul* to redelegate this role. But he also specifies that *Paul* is allowed to redelegate the role *Head of department* only if he is in vacation.

First, to delegate the role *Head of department* to *Paul*, *Bob* must proceed similarly as in *Example 3*. He inserts in the view *role_assignment* the following object:

Grantor: Bob,
Assignee: Paul,
Assignment: Head_dep.

According to $\boldsymbol{Rule}_{RA}$, the following fact is created:

$$Empower(paul, head\_dep)$$

Then, to give *Paul* the permission to redelegate this role, *Bob* must create and insert the following licence *bob_to_paul* in the view *grant_option*:

Grantor: Bob,
Grantee: Paul,
Privilege: Delegate,
Target: Head_view,
Context: Vacation(Alex) & Vacation(Paul),
Step: 2.

Note that *Bob* is not allowed to insert a licence in the view *grant_option* that does not satisfy the context *valid_ redelegation*($alex\_to\_bob_2$). Thus, he must consider the following conditions:

- the privilege must be a delegation action (not a transfer action),
- the target must satisfy the definition of view *head_view*,
- the context vacation(alex) must be included in the context of the inserted object.
- the step must be lower than 3.

According to $\boldsymbol{Rule}_{LA'}$ and $\boldsymbol{Rule}_{Gop_2}$, these permissions are created respectively:

$$
\begin{aligned}
&Permission(paul, delegate, head\_view, vacation(alex)\\
&\qquad \& vacation(paul)).\\
&Permission(paul, delegate, grant\_option,\\
&\qquad vacation(alex) \,\&\, vacation(paul)\\
&\qquad \& valid\_redelegation(bob\_to\_paul)).
\end{aligned}
$$

As we have shown in this section, thanks to the *grant_ option* view, we made a distinction between the delegation of a simple right *R* and the delegation of the right to delegate *R*. So, we can deal with multi-step and agent-based delegation simply and securely. This is not the case of the *With Grant Option* in relational database systems [18], which is used only with the *Grant* command. Thus, one cannot delegate the right to delegate *R* without delegating *R*, and therefore the agent-acted delegation is not supported. Moreover, using the *With Grant Option*, one cannot specify contextual permissions or delegation constraints as in our model. For instance, it is not possible to specify conditions to limit the scope of delegation, such as the delegation step, the grantee (i.e. users-list to which delegation is allowed), or the context of the delegation (e.g. vacation).

Managing the delegation of delegation right

Managing this view is completely similar to the managing of the *licence_delegation* view, this is defined by facts having the following form:

$Permission(Gr, delegate, grant\_option, C).$
$Permission(Gr, delegate, sub\_grant\_option, C).$

where *sub_grant_option* is a sub-view of *grant_option* defined as follows:

$Use(LG, sub\_grant\_option)$:-
    $Use(LG, grant\_option), Licence\_Condition.$

This means that the administrator can specify different types of conditions concerning all the licence attributes, namely the grantor, the grantee, the privilege, the target, the context and/or the step of delegation. Moreover, as we have explained in Sect. 3.3.1, he can specify extra conditions thanks to contexts. In the following section, we explain how to use contexts to manage delegation requirements.

3.4 Contextual delegation

As we have mentioned earlier, there are 5 kinds of contexts defined in OrBAC [12]. Temporal: depends on the system clock; Spatial: depends on subject location; Provisional: requires that some previous actions must be executed; Prerequisite: depends on specific information stored in a database; and User-declared: the activation of this context is decided by some authorized user.

These contexts are very useful in delegation, since we can express complex conditions to specify delegation constraints. As we have shown in previous sections, these conditions can either be specified by the administrator to manage the delegation policy like in examples ***2***, ***4*** and ***6***, or by the grantor to limit the scope of delegated rights like in examples ***8*** and ***9***. We give here more examples to deal with delegation requirements, namely multiple delegation and temporary delegation.

Example 10 **Multiple delegation** This characteristic refers to the number of grantees to whom a grantor can delegate the same right (permission or role) at any given time. This number, which we call N_m, is fixed by the administrator by using a provisional context. We can specify the context *multiple_Licence_Deleg* to compare N_m with the delegation number concerning the same grantor and the same permission:

$Hold(S, A, L, multiple\_Licence\_Deleg(N_m))$:-
    $Use(L, licence\_delegation),$
    $Count(L', (Use(L', licence\_delegation),$
    $Equiv\_Licences(L, L')), N'_m), N'_m <= N_m.$

We assume that *Count(V, p(V),N)* is a predicate that count the set of instances of variable *V* that satisfies predicate *p(V)*. *N* represents the result of the count predicate.

Note that we consider licences *L* and *L'* are equivalent if they are delegated by the same grantor and concern the same right:

$Equiv\_Licences(L, L')$:-
    $Grantor(L, U), Grantor(L', U),$
    $(Sub\_Licence(L, L'); Sub\_Licence(L', L)).$

To explain this, let us consider the delegation constraint ***6-b*** of our example. If we assume that *Paul* delegates the access to his student's marks to his assistant *Bill*, then he does not have the permission to delegate this right to another user. For instance, he cannot delegate the permission to access to his master student's marks, since this right is a sub-licence of the first one.

Notice that when the *multiple_Licence_Deleg* context is not used, the number of permissions a subject can delegate is not restricted. So this subject can delegate as many licences he/she wants.

Similarly, we can define another provisional context to limit the number of grantees to whom a grantor can delegate the same role at any given time:

$Hold(S, A, RD, multiple\_Role\_Deleg(N_m))$:-
    $Use(RD, role\_delegation),$
    $Count(RD', (Use(RD', role\_delegation),$
    $Equiv\_RD(RD, RD')), N'_m),$
    $N'_m <= N_m.$
$Equiv\_RD(RD, RD')$:-
    $Grantor(RD, U), Grantor(RD', U),$
    $Assignment(RD, Role),$
    $Assignment(RD', Role).$

This context is useful to express the delegation constraint ***7-b*** of our example, where role *chef_dep* must be delegated to only one professor at a given time.

Example 11 **Temporary delegation** Thanks to the use of the temporal context, we can easily deal with this characteristic. The grantor can specify that the delegated permission is authorized only at a given time, after or before a given time, or during a given time interval. The temporal conditions may correspond to a day of the week, or to a time of the day, etc.

As we have described in examples ***8*** and ***9***, the grantor can include a temporal condition in the context of the delegated licence, for instance during his/her vacation. Thus, the delegated permission applies only if the context holds. If the temporal context is not used, the delegation is permanent.

Let us consider constraint ***8*** of our example and assume that *Paul* delegates to his assistant *Bill* a temporary permission to proof-read his examinations that only applies during the examination period:

Grantor: Paul,
Grantee: Bill,
Privilege: Access,
Target: Paul_stud_mark,
Context: Exam_period.

where the temporal context *exam_period* is defined as follows:

$$Hold(U, A, O, exam\_period)\text{:-} \\ Hold(U, A, O, after\_date(19/03/2008)\& \\ before\_date(19/04/2008).$$

We consider that temporary delegation is not restricted to temporal contexts. It may also be modelled using the other types of contexts, since the delegation only applies temporarily, i.e. when the context is active. For instance, if we consider constraint ***9*** of our example then $Paul$ can specify that the delegation is valid only if $Bill$ is located in his office. This is a spatial context, and the delegation applies temporarily according to the physical location of the grantee:

Grantor: Paul,
Grantee: Bill,
Privilege: Access,
Target: Paul_stud_mark,
Context: Location(paul_office).

where the spatial context *location* is defined as follows:

$$Hold(S, \alpha, O, location(So))\text{:-} \\ Is\_located(S, So).$$

where *Is_located (S, So)* is an application-dependent predicate meaning that the subject *S* is located in the area of spatial object *So*.

Note that this kind of conditions is not supported by other models that only support temporal constraints. In fact, the grantor can specify constraints to the delegated permissions using contexts. These constraints concern the grantee's characteristics (e.g. role, location, affiliation, actions, status) or also the time, the circumstance, etc. Hence, in our model, we extend the notion of temporary delegations to the more general notion of contextual delegations.

3.5 Managing delegation privileges

As we have shown in previous sections, our delegation model is completely homogeneous with the remainder of the extended RBAC model. The approach we suggest is to define delegation views that are similar to administrative views.

Thanks to this approach, we have proposed a delegation model without adding new types of roles, permissions or relationships to deal with delegation requirements. Delegation privileges, namely permissions and roles are assigned to users just like administrative privileges. This is not the case in existing models such as RBDM [4], PBDM [33], RDM2000 [32], which distinguish between regular and delegated rights (roles or permissions). For instance, in [23], authors define six different layers of roles and seven partitions of permissions to deal with partial delegation, permission level delegation and restricted inheritance.

This is a complex task to manage, since there are many different types of roles and permissions. Moreover, this implies that it is necessary to define new policies to manage hierarchies and separation of entities for these additional roles and permissions, like in [20,23]. More details about existing models will be given in Sect. 7.

In our model, the delegation of rights by a grantor, either permission or role, will create, respectively, a *Permission* or an *Empower* just like the assignment of rights by the administrator. This is very useful to simplify the administrator task. Indeed, managing delegation privileges does not necessitate the specification of a separate security policy dedicated to them, since there is no distinction between the regular roles and the delegation roles, and there is no specific permissions for the delegation tasks such as *can_delegate* and *can_revoke* in RBAC-based delegation models. Notice that the *can_delegate* relationship actually combines two different concepts, namely a permission and an action of delegation. As suggested in this paper, we claim that it is more appropriate to separately model these two concepts. This provides higher flexibility, and it is also easier to extend the model by considering new types of delegation. The concept of context also provides higher expressivity and finer grained delegation than other already defined delegation models.

3.5.1 Delegation hierarchy and inheritance

Inheritance rules presented in Sect. 2.2 apply also to delegation privileges, since there is no distinction between the regular roles/permissions and the delegation roles/permissions. This means that, according to $\boldsymbol{Rule}_{Sub_R}$, if a grantee is empowered by delegation (or by transfer) to a given role, then he/she inherits automatically the permissions of his/her sub-roles. Moreover, in the case of transfer, the grantor loses his/her role and permissions of his/her sub-roles as well, since he/she is no longer empowered in the transferred role.

Note that if the grantor wants to delegate (or transfer) a role without sub-role inheritance, he/she may delegate (or transfer) all the role permissions instead of delegating the role. Hence, in the case of transfer, the grantor loses only the permissions of the role that he/she transferred, but not the permissions of his/her sub-roles.

$\boldsymbol{Rule}_{Sub_V}$ and $\boldsymbol{Rule}_{Sub_A}$ specify inheritance over views and activities. This simplifies the delegation task since the grantor can delegate in one time a set of permissions. For instance, when professor *Paul* delegates to his assistant

Bill the access to his student's marks, this means that *Bill* is allowed to perform the activity *access* on the view *paul_stud_marks*, but also he is allowed to perform sub-activities, namely *read, write, delete*, on all files or directories containing the student's marks of *Paul*.

This also simplifies the administrator task, since if he/she gives a permission to a grantor to delegate (or transfer) a given right, then the grantor is allowed to delegate (or transfer) also its sub-rights.

However, this is not the case for role delegation (or transfer). This means that if the grantor is allowed to delegate a role, then he/she is not automatically allowed to delegate its sub-roles. For this purpose, the administrator must specify explicitly this permission. For instance, if he/she wants to give role *Head_dep* the permission to delegate its role and its sub-roles in some context C, then the administrator must specify this condition as follows:

$$Permission(head\_dep, delegate, sub\_role\_deleg(head\_dep), C).$$

where sub-view sub_role_deleg is defined as follows:

$$\begin{aligned} &Use(RD, sub\_role\_deleg(R)):\\ &\quad Use(RD, role\_delegation),\\ &\quad Assignment(RD, R'),\\ &\quad Sub\_role(R, R'). \end{aligned}$$

3.5.2 Global constraints

In our model, global constraints specified by the administrator in the security policy apply automatically to delegation privileges. For instance, if we have the following separation of entities:

$$Separated\_role(professor, secretary).$$

And we assume that professor *Bob* delegates his role to *Mary*, the secretary. We will have the following facts:

$$\begin{aligned} &Empower(mary, secretary).\\ &Empower(mary, professor). \end{aligned}$$

According to ***Rule***$_{Cst_R}$, the global constraint is violated. Hence, the delegation is rejected.

Although, in our model, there is no distinction between administrative and delegation privileges, there is no confusion between these two privilege types. This is possible since administrative privileges are derived from objects belonging to administrative views, whereas delegation privileges are derived from objects belonging to delegation views.

For instance, we assume that the administrator and a given grantor Gr assign the same permission P to the same user U. When Gr revokes the permission he/she has delegated to U, only the object belonging to the delegation view *licence_delegation* is deleted. But the permission assigned by the administrator is not revoked.

3.5.3 Conflict management

In our model, the delegation concerns only positive privileges, since we consider it is not meaningful to delegate prohibitions. Hence, during delegation process, there is no creation of new prohibitions by users. Except in the case of licence transfer, when a prohibition is automatically created for the grantor. This prohibition is associated with the highest priority level. Hence, the conflict that may occur with the other permissions is solved.

Conflict may occur only between the delegated permissions and the prohibitions specified by the administrator. Since delegation is considered as an exception to the security policy, in our model delegated permissions are associated with higher priority than general rules (see Sect. 2.4). For instance, let us assume that the administrator specifies that role *Assistant* is prohibited to access the student's marks:

$$Prohibition(assistant, access, stud\_mark, nominal). \tag{R_1}$$

And let us assume that professor $Paul$ delegates the following permission to his assistant $Bill$:

$$Permission(bill, access, paul\_stud\_mark, nominal). \tag{R_2}$$

In our model, rule R_2 will be assigned higher priority than R_1, since R_2 is an exception of R_1. Hence, the delegation will take precedence.

4 Revocation

In our model, revocation can be performed automatically when the delegation context is no longer active, or manually by an authorized user. User revocation is performed by deleting objects from the delegation views. The effect of the revocation on the other delegations depends on the revoker needs. He/she can choose, for instance, to revoke the whole delegation chain (cascade revocation) or to revoke all other delegations associated with the same grantee (strong revocation). To deal with these features, we give in the following a sketch of our approach to manage revocation(see [9] for more details).

4.1 Notations and definitions

We denote L, R and LG three delegation objects as follows: $L \in licence\_delegation$, $R \in role\_delegation$ and $LG \in grant\_option$.

Let O_{SD} be the set of simple delegations SD (L or R), O_{LG} the set of delegation licences LG and $O_D = O_{SD} \cup O_{LG}$ the set of all delegations D.

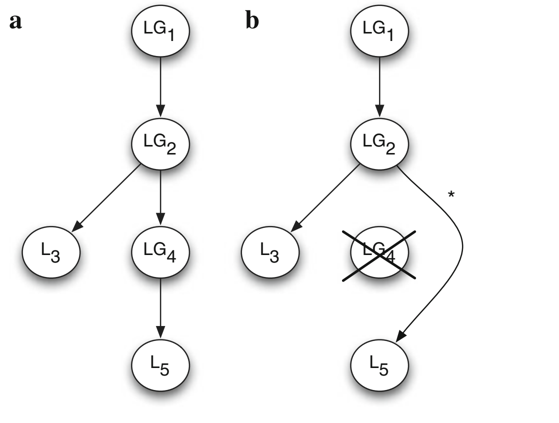


Fig. 3 **a** Delegation chain DC(LG$_1$). **b** Delegation chain DC*(LG_1)

Definition 1 Derivation relation

We distinguish between two derivation relations according to whether the type of derived delegation is a simple or a delegation licence.

- $\forall LG, LG' \in O_{LG}$, LG is **derived** from LG', if:
  - the grantor of the licence LG is the grantee of the licence LG',
  - the licence LG is a sub-licence of LG',
  - the delegation step of LG is lower than the delegation step of LG'.
- $\forall SD \in O_{SD}$, $LG \in O_{LG}$, SD is **derived** from LG, if:
  - the grantor of the delegation SD is the grantee of the licence LG,
  - The delegation SD corresponds to the target definition of LG (i.e. *Use(SD, Target(LG))*).

Definition 2 Delegation chain

- For each delegation licence $LG \in O_{LG}$ we can generate a delegation chain, which we call $DC(LG)$. A delegation chain is represented by a directed graph (see Fig. 3a). The nodes contain delegations $D \in O_D$, and we denote $N(D)$ the node containing delegation D. There is an arc from node $N(D_1)$ to node $N(D_2)$, if D_2 is derived from D_1. A node containing a simple delegation $SD \in O_{SD}$ is always a leaf of the graph.
- A node is rooted if it contains a delegation that cannot be derived from any other licences. A delegation chain of a delegation D is rooted if D is rooted.
- When a node N_i is deleted (i.e. the delegation contained in this node is revoked), a special arc labelled with a * is used to connect nodes N_{i-1} to N_{i+1} (see Fig. 3b). In addition, we denote $DC*(D)$ the delegation chain $DC(D)$ that includes labelled arcs.

Note that the delegation chain DC* is used to ensure that every delegation has a path that links it to the licences from which it is indirectly derived, even if some licences belonging to the delegation chain are removed. For instance, if we consider the delegation chain given in Fig. 3b, then $DC(LG_1)$= $\{LG_2, L_3\}$ and $DC*(LG_1)$ =$\{LG_2, L_3, L_5\}$.

Definition 3 Dependency

A delegation D depends exclusively on a user U if there is no rooted delegation chain DC such that $D \in DC$ and $\forall D' \in DC, Grantor(D') \neq U$.

We assume that *Dependent(D, U)* is a predicate meaning that delegation D depends exclusively on user U.

Definition 4 Simple revocation

This is the simplest revocation scheme. The revocation involves deleting objects from the delegation view and does not affect the other delegations. We define the request to revoke a delegation as follows:

```
Request(U, revoke, D) then
  forall ($D_i \in$ Derived_deleg(D) and
   $D_j \in$ Parent_deleg(D)) do
    Add a labeled arc from N($D_j$) to N($D_i$),
    Remove(D).
end
```

We assume that *Derived_deleg(D)* is the set of all delegations D' such that there is an arc (labelled or not labelled) from node $N(D)$ to node $N(D')$ and *Parent_deleg(D)* is the set of delegations D'' such that there is an arc (labelled or not labelled) from node $N(D'')$ to node $N(D)$.

Definition 5 Cascade revocation

This involves the revocation of all the delegations belonging to the delegation chain. But this revocation should not affect the ones belonging to other delegation chains (DC). The reason is that if the grantor of delegation, D has received the permission to delegate it from two or more different delegations, then if one of these delegations is revoked, the grantor still has the right to delegate D (an illustrative example is given below). In our model, cascade revocation is defined as follows:

```
Request(U, Cascade_revoke, D) then
  Request(U, revoke, D),
  forall $D_i \in$ Derived_deleg(D) do
    if $D_i \notin DC(LG), LG \in O_{LG}$ then
      Request(U, Cascade_revoke, $D_i$).
  end
end
```

Example 12 We consider the example shown in Fig. 4, and we assume that LG_1 is revoked with the cascade option. On the first pass, licence LG_1 is removed. On the second pass,

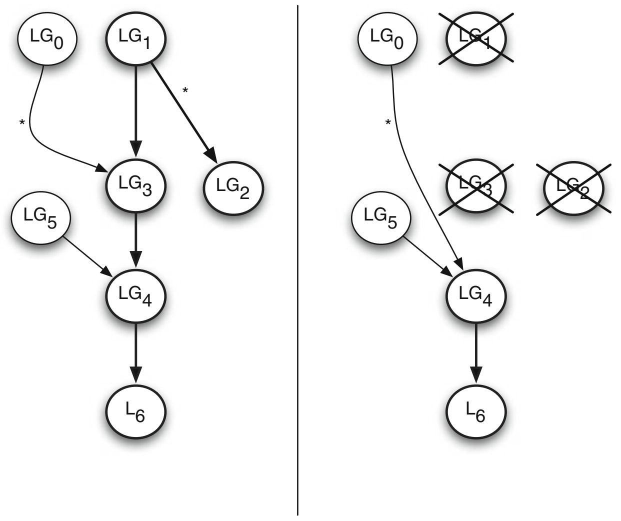


Fig. 4 Cascade revocation

LG_2 and LG_3 are revoked, and a labelled arc is added to connect node LG_0 to LG_4. Note that LG_3 is revoked because it belongs to $DC*(LG_0)$ and not to $DC(LG_0)$. Finally, the cascade revocation process is stopped because LG_4 belongs to $DC(LG_5)$.

The revocation described so far is a weak revocation. This means that the revocation of a delegation D will only affect D (in the case of simple revocation) or the delegations belonging to its delegation chain (in the case of cascade revocation). By contrast, a strong revocation will affect the other delegations associated with the same grantee.

Definition 6 Strong revocation
The strong revocation of delegation D means that all the delegations equal to D (or are a sub-right of D) that depend on the revoker and associated with the same grantee must be revoked. This is defined as follows:

```
Request(U, Strong_revoke, D) **then**
Request(U, revoke, D),
**forall** $D_i \in O_D$ **do**
    **if** *(Grantee($D_i$) = Grantee(D)* **and** *Sub_right($D_i$,D)*
    **and** *Dependent($D_i$,U))* **then**
        Request(U, revoke, $D_i$).
    **end**
**end**
```

We assume that the predicate *Sub_right(D, D')* is defined as follows, according to whether the type of right is a licence delegation or a role delegation:

$Sub\_right(D, D')$:-
$Use(D, licence\_delegation)$,
$Use(D', licence\_delegation)$,
$Sub\_licence(D, D')$.

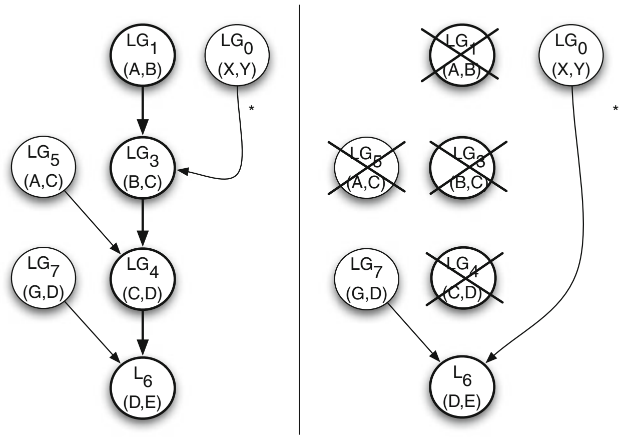


Fig. 5 Strong-cascade revocation

$Sub\_right(D, D')$:-
$Use(D, role\_delegation)$,
$Use(D', role\_delegation)$,
$Assignment(D, R)$, $Assignment(D', R')$,
$Sub\_role(R, R')$.

Definition 7 Strong-Cascade revocation
We consider the strong-cascade revocation of a delegation D as a strong revocation of all delegations belonging to the delegation chain of D. This is defined as follows:

```
Request(U, Strong_Cascade_revoke, D) **then**
Request(U, Strong_revoke, D)
**forall** $D_i \in derived\_deleg(D)$ **do**
    **if** $D_i \notin DC(LG)$, $LG \in O_{LG}$ **then**
        Request(U, Strong_Cascade_revoke, $D_i$)
    **end**
**end**
```

Example 13 We consider the example shown in Fig. 5, where we represent in each node the delegated licence, the grantor and the grantee of this licence. We assume that licence LG_5 is a sub-licence of LG_3 and LG_7 is a sub-licence of LG_4.

If A the grantor of licence LG_1 revokes this licence with the strong-cascade option, then the whole delegation chain of LG_1 will be revoked with the strong cascade option as well. On the first pass, licence LG_1 is revoked by A. On the second pass, licence LG_3 is revoked and also licence LG_5 since it is a sub_licence of LG_3 and it depends on A. On the third pass, LG_4 is revoked but not LG_7 because it is independent of A. L_6 is not revoked because it belongs to $DC(LG_7)$.

4.2 Managing revocation

To perform a revocation, users must have the permission to delete objects from the delegation views. This permission may be specified by the administrator just like delegation.

He/she must specify which user is permitted to revoke objects in the delegation views. For this purpose, we do not use specific functions to define the right to revoke such as *can_revoke*, *can_revokeGD*, *can_revokeGI* and *can_u2u_revokeGD* like in [5,21,32]. We have simply to specify revocation constraints using contexts. In the following, we give some examples to illustrate how to use them to specify revocation constraints.

Example 14 **Grant Dependent** In the case of Grant-Dependent revocation (GD), only the grantor is allowed to revoke the delegated licence or role. To deal with this aspect, we define the prerequisite contexts GD_L and GD_R for the GD licence revocation and the GD role, respectively:

$Hold(U, revoke, L, GD_L)$:-
  $Use(L, licence\_delegation), Grantor(L, U)$.
$Hold(U, revoke, RD, GD_R)$:-
  $Use(RD, role\_delegation), Grantor(RD, U)$.

For instance, if we consider the revocation constraints ***11*** of our example, then the administrator can specify that all users are authorized to revoke their licence delegation or role delegation as follows:

$Permission(default\_Role, revoke, licence\_delegation, GD_L)$.
$Permission(default\_Role, revoke, role\_delegation, GD_R)$.

Similarly, we can associate contexts with the Grant-Dependent revocation of licence and role transfer.

Example 15 **Grant Independent** We can specify different contexts to deal with this aspect according to the administrator needs. For instance, we consider the revocation constraint ***12*** of our example, where role professor is allowed to revoke role delegation that depends on him. To specify this constraint, we may specify context *Ancestor Dependent* ad_R as follows:

$Hold(U, revoke, RD, ad_R)$:-
  $Use(RD, role\_delegation), Dependent(RD, U)$.

Similarly, we may specify context ad_L to allow users to revoke permission delegation that depends on them:

$Hold(Org, U, revoke, L, ad_L)$:-
  $(Use(Org, L, licence\_delegation)$;
  $Use(Org, L, grant\_option)), Dependent(L, U)$.

We may also consider different revocation constraints, such as:

– Users are permitted to revoke delegated roles if they are empowered in a role that is hierarchically superior:

$Hold(U, revoke, RD, GI_{Sub})$:-
  $Use(RD, role\_delegation), Assignment(RD, R)$,
  $Empower(U, R'), Sub\_role(R', R)$.

This context can be used to specify constraint ***13*** of our example, where the director is allowed to revoke role delegation of the other users:

$Permission(director, revoke, role\_delegation, GI_{Sub})$.

– Users are permitted to revoke a privilege (permission or role) if they are assigned to this privilege. For this purpose, we may define the following contexts:

$Hold(U, revoke, RD, GI_R)$:-
  $Use(RD, role\_delegation)$,
  $Assignment(RD, R), Empower(U, R)$.
$Hold(U, revoke, L, GI_L)$:-
  $Use(L, licence\_delegation)$,
  $Privilege(L, P), Target(L, T), Context(L, C)$,
  $Permission(U, P, T, C)$.

– Users are permitted to revoke a privilege if they are permitted to delegate it:

$Hold(U, revoke, RD, GI_{DR})$:-
  $Use(RD, role\_delegation)$,
  $Is\_Permitted(U, delegate, RD)$.
$Hold(U, revoke, L, GI_{DL})$:
  $Use(L, licence\_delegation)$,
  $Is\_Permitted(U, delegate, L)$.

Moreover, we can define other conditions that may concern the characteristics of the grantor or the grantee, the previous actions, the delegated right (i.e. the role, the target, the privilege), the time, the circumstances (e.g. urgency), etc. For instance, we consider the revocation constraint ***14*** of our example, and we specify that role *Director* is allowed to revoke transferred roles only if the grantor of this transfer is not in vacation:

$Hold(U, revoke, RD, GI_{Tr})$:-
  $Use(RD, role\_transfer), grantor(RD, Gr)$,
  $\neg In\_vacation(Gr)$.
$Permission(director, revoke, role\_transfer, GI_{Tr})$.

We can also define that a user U_1 can revoke a grantee U_2 if U_2 has performed a given action A on a given object O. We may also specify that U_1 is authorized to revoke U_2, if U_2 is the assistant of U_1, or is associated with the same department as U_1. This kind of conditions is not supported by the existing models since they only specify revocation constraints on the role membership of the grantor or the grantee.

5 Decidability and complexity

Our model is based on the OrBAC formalism which is compatible with first-order logic and more precisely with Datalog [30]. Datalog ensures a decidable and tractable theory.

Datalog programs must only include both defined and safe rules. A rule is defined if every variable that appears in the conclusion also appears in the premise. A rule is safe if it only provides means to derive a finite set of new facts. In a pure Datalog program, rules do not contain any negative literal. Pure Datalog guarantees that any access control policy will be decidable in polynomial time. However, pure Datalog expressivity is very restricted.

In Datalog$^\neg$, the negation restriction is relaxed. Negative literals are allowed but rules must be stratified [30]. A stratified Datalog$^\neg$ program is computable in polynomial time.

The definition of security policies using the OrBAC model obeys the Datalog$^\neg$ restriction except the definition of contexts through the *hold* predicate. More precisely, the security rules correspond to ground close facts specified using the *permission*, *prohibition* predicates. Specifications of predicates *empower*, *use* and *consider* correspond also to facts or rules that respect the Datalog$^\neg$ restriction.

By contrast, the definition of contexts does not correspond to Datalog$^\neg$ restriction for the following reasons:

– These rules are not always safe. For instance, temporal context definitions such as *after-time(t)* may derive an infinite set of new facts when time t is not fixed.
– These rules are not always defined. For instance, in temporal context definitions, subjects, actions and objects are respectively universally quantified over the set S, A and O but are not further constrained in the premises of the rule. This may lead to evaluate large Cartesian products which is not efficient.

To solve these problems, it is proposed firstly to restrict the theory so that only *relevant* contexts are evaluated. A context is relevant if it appears in the definition of a security rule. Secondly, a relevant context is always fully instantiated. Finally, it is proposed to pre-compute the evaluation of the *Empower*, *Use* and *Consider* predicates using a bottom-up strategy. Then, the evaluation of queries is completed using the top-down strategy as defined in the SGL algorithm [29]. This hybrid strategy guarantees the decidability of query evaluation in the OrBAC model and its termination in polynomial time.

We can now prove the following theorems:

Theorem 1 *The delegation chains are computable in polynomial time.*

Proof The delegation chain is based on the OrBAC model and its self-administration model. As we mentioned, policies associated with both of them can be expressed as recursive rules corresponding to a stratified Datalog program; the delegation chains are then obtained by computing a fixed point which is tractable in polynomial time. □

Theorem 2 *The revocation requests are computable in polynomial time.*

Proof A request to revoke a licence requires a recursive search in the delegation chain of this licence; therefore, using Theorem 1, it is computable in polynomial time. □

6 The MotOrBAC tool

MotOrBAC is a tool developed in Telecom Bretagne to implement the OrBAC model and its administration model Ad-OrBAC. It provides a user-friendly interface to specify and manage the security policy. We have integrated our proposed model into MotOrBAC to manage delegation and revocation features. More details about this tool are given in [22], and it is available in the MotOrBAC web site [28].

MotOrBAC architecture is given in Fig. 6.

The OrBAC application programming interface (API) is used to manage the policies displayed in the graphical user interface (GUI). The API uses the Jena Java library [19] to represent an OrBAC policy as an RDF graph. It can be used to load MotOrBAC RDF policies and interpret them. Jena features an inference engine which is used by the OrBAC API to infer the concrete policies. When an OrBAC RDF policy is loaded by the API, the concrete policy can be inferred and stored in memory. An instance of the $OrbacPolicy$ Java class which encapsulates an OrBAC policy uses a cache of concrete security rules to enhance the performances when the policy is queried. Contexts are evaluated upon a query. This feature is actually used in the MotOrBAC simulation tool to show the contexts state. The contexts implementation can be easily extended in order to interface the API with other applications and add new types of contexts.

Moreover, it is possible to integrate the OrBAC Java API into a Java application without modifying the application source code. This is done using Aspect Oriented Programming (AOP) to separate security requirements from other concerns related to the application. Using AspectJ the security requirements are weaved with the OrBAC API into the application code. The API can also be used to create OrBAC policies, it can be for instance integrated into a web server to communicate with a web browser which could run a web interface to create and/or administrate OrBAC policies. We give in the following the basic functions supported by this tool, namely administration functions performed by the administrator(s) to manage the security policy (using the MotOrBAC GUI), and delegation functions performed by

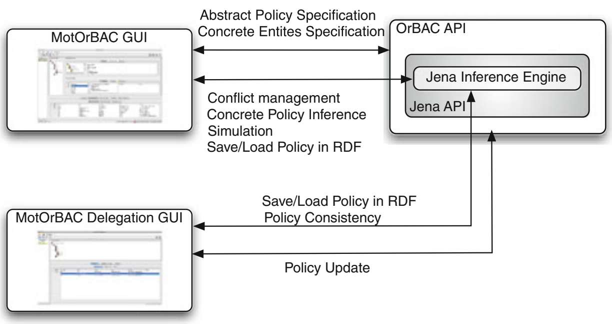
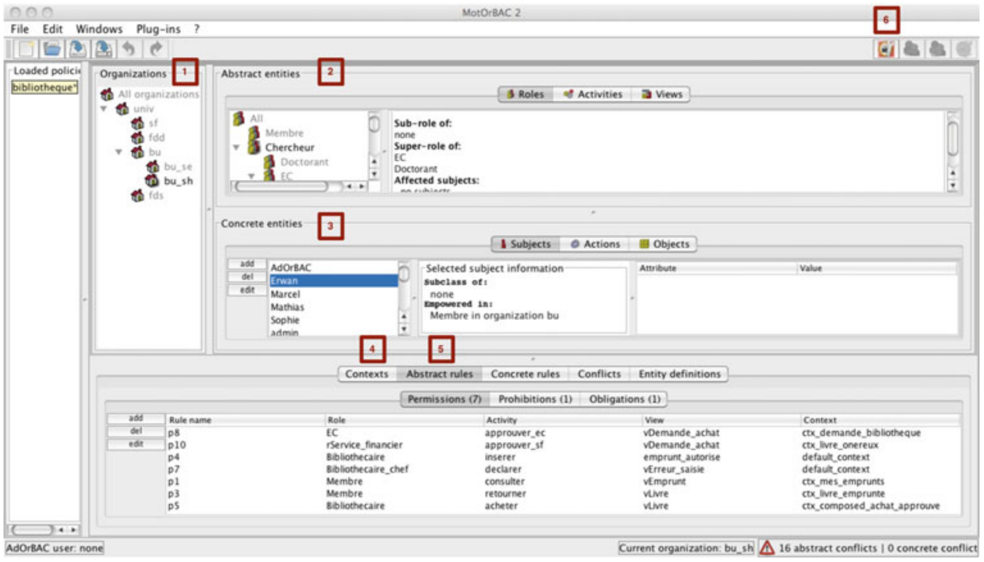


Fig. 6 The MotOrBAC architecture



Sub-role of:
none
Super-role of:
EC
Doctorant
Affected subjects:

Selected subject information
Subclass of:
none
Empowered in:
Membre in organization bu

Rule name	Role	Activity	View	Context
p8	EC	approuver_ec	vDemande_achat	ctx_demande_bibliotheque
p10	rService_financier	approuver_sf	vDemande_achat	ctx_livre_onereux
p4	Bibliothecaire	inserer	emprunt_autorise	default_context
p7	Bibliothecaire_chef	declarer	vErreur_saisie	default_context
p1	Membre	consulter	vEmprunt	ctx_mes_emprunts
p3	Membre	retourner	vLivre	ctx_livre_emprunte
p5	Bibliothecaire	acheter	vLivre	ctx_composed_achat_approuve

AdOrBAC user: none

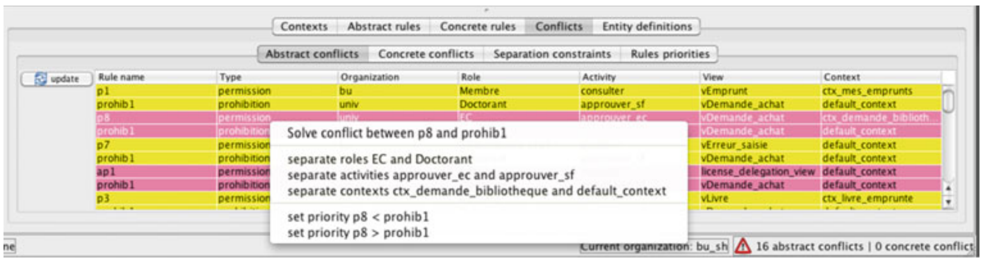
Current organization: bu_sh | 16 abstract conflicts | 0 concrete conflict

Fig. 7 MotOrBAC GUI

regular users to delegate or revoke their rights (using the MotOrBAC Delegation GUI).

Administration functions

- Edit policies: the administrator can create the abstract entities he/she needs, namely, organizations (see part 1 in Fig. 7), roles, activities, views (part 2, Fig. 7) and the relationships between these entities, namely hierarchies. He/she can also specify contexts using part 4 of Fig. 7. Then the administrator can specify the abstract security policies for these abstract entities (i.e. permissions and prohibitions using part 5 of Fig. 7). Given entities hierarchies the tool applies automatically the inheritance rules, presented in Sect. 2.2, using the Jena engine.
- Conflict management: MotOrBAC helps to detect abstract conflicts using the conflict management strategy discussed in Sect. 2.4. The detected abstract conflicts are listed in the GUI and separation constraints as well as rule priorities can easily be added through a contextual menu to resolve these conflicts (see Fig. 8).
- Policy simulation: after having specified concrete entities, subjects, actions and objects (part 3, Fig. 7), the concrete policy can be inferred using the Jena engine (i.e. actual permissions and prohibitions). Figure 9 shows the simulation window which lists the concrete security rules (a different color is used for each rule type). Light-colored entries shows concrete security rules for which the associated context is inactive. Note that concrete conflicts can also be displayed in MotOrBAC, but the administrator is not allowed to solve them at the concrete level.
- Administrative rights management: the administrative rights of a subject or a role can be specified in order to decentralize the policy administration. Therefore, when the AdOrBAC function is activated in MotOrBAC (part 6 in Fig. 7), after the user has authenticated himself/herself, he/she can edit the OrBAC security policy accordingly to the AdOrBAC policy. If an unauthorized operation is attempted, the policy is not modified and the user is informed. Since the OrBAC model is self-administrated, a MotOrBAC policy file contains both the OrBAC policy and its associated AdOrBAC policy.
- Delegation rights management: similarly to the administrative rights management, MotOrBAC allows the



Rule name	Type	Organization	Role	Activity	View	Context
p1	permission	bu	Membre	consulter	vEmprunt	ctx_mes_emprunts
prohib1	prohibition	univ	Doctorant	approuver_sf	vDemande_achat	default_context
p8	permission	univ	EC	approuver_ec	vDemande_achat	ctx_demande_biblioth...
prohib1	prohibition				vDemande_achat	default_context
p7	permission				vErreur_saisie	default_context
prohib1	prohibition				vDemande_achat	default_context
ap1	permission				license_delegation_view	default_context
prohib1	prohibition				vDemande_achat	default_context
p3	permission				vLivre	ctx_livre_emprunte

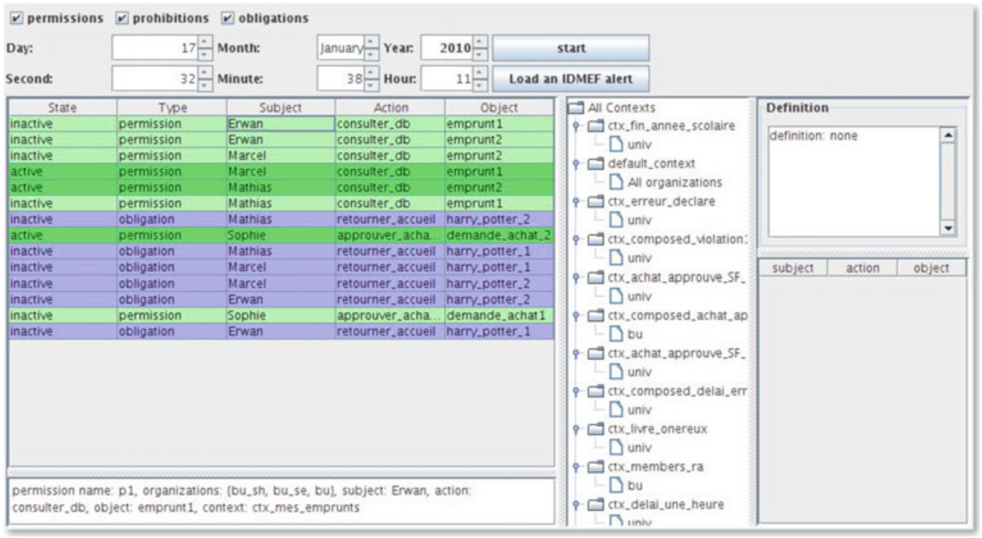
Solve conflict between p8 and prohib1

separate roles EC and Doctorant
separate activities approuver_ec and approuver_sf
separate contexts ctx_demande_bibliotheque and default_context

set priority p8 < prohib1
set priority p8 > prohib1

Current organization: bu_sh | 16 abstract conflicts | 0 concrete conflict

Fig. 8 Abstract conflict tab



State	Type	Subject	Action	Object
inactive	permission	Erwan	consulter_db	emprunt1
inactive	permission	Erwan	consulter_db	emprunt2
inactive	permission	Marcel	consulter_db	emprunt2
active	permission	Marcel	consulter_db	emprunt1
active	permission	Mathias	consulter_db	emprunt2
inactive	permission	Mathias	consulter_db	emprunt1
inactive	obligation	Mathias	retourner_accueil	harry_potter_2
active	permission	Sophie	approuver_acha...	demande_achat_2
inactive	obligation	Mathias	retourner_accueil	harry_potter_1
inactive	obligation	Marcel	retourner_accueil	harry_potter_1
inactive	obligation	Marcel	retourner_accueil	harry_potter_2
inactive	obligation	Erwan	retourner_accueil	harry_potter_2
inactive	permission	Sophie	approuver_acha...	demande_achat1
inactive	obligation	Erwan	retourner_accueil	harry_potter_1

permission name: p1, organizations: {bu_sh, bu_se, bu}, subject: Erwan, action: consulter_db, object: emprunt1, context: ctx_mes_emprunts

definition: none

Fig. 9 Concrete policy simulation window

administrator to specify the delegation policy. Therefore, authorized users can delegate or revoke their rights using the dedicated Delegation GUI given in Fig. 10.

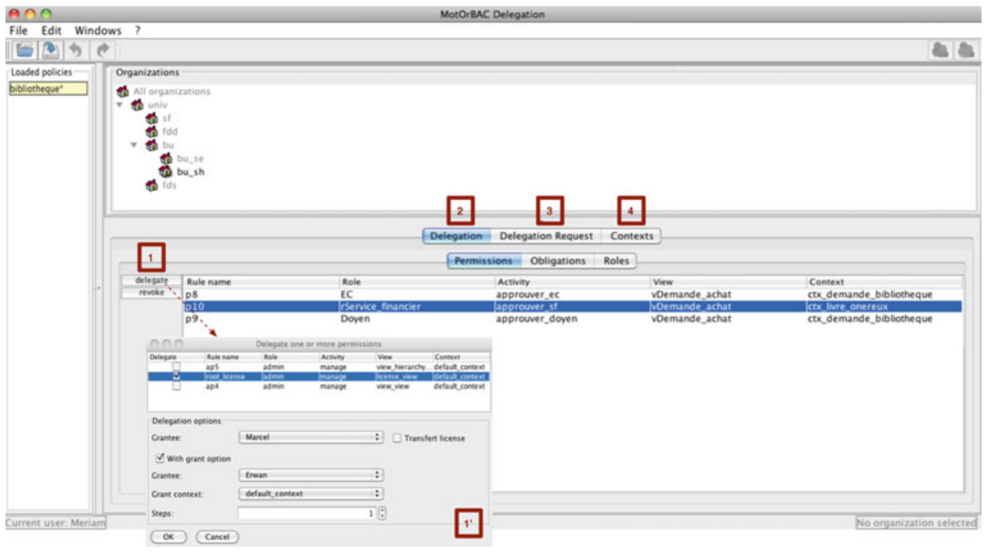
Delegation functions The Delegation GUI (see Fig. 10) is quite similar to the MotOrBAC GUI and is based on the same OrBAC API to manage delegation and revocation. But in the Delegation GUI, some features are removed, namely not allowed administration features such as the creation of new entities or conflict management, and other features specific to delegation are added, such as transfer, multi-step delegation or revocation features. More details about Delegation GUI are given in [7].

According to the delegation policy, the Delegation GUI allows authorized users:

– to edit the security policy related to them: permissions and roles that they delegated to other users and also those they have been assigned through delegation (part 2, Fig. 10),
– to delegate and to revoke their rights using delegation and revocation features that they are authorized to perform such as transfer, delegation with grant-option, revocation with cascade or strong option. (parts 1 and 1', Fig. 10),
– to specify delegation contexts (part 4, Fig. 10),
– to request other users to delegate to them a given right or to revoke an existing delegation (part 3, Fig. 10). As we proposed in [7], a delegation protocol can be used to manage users communication and to negotiate the delegation or the revocation of rights. We plan to further investigate this aspect in future work.

7 Discussion and related work

During the last years, several papers have been published dealing with delegation in Role-Based Access Control models. These works distinguish between administration and delegation process and propose to extend the RBAC model by adding separate components, such as new types of roles, permissions, relationships. Hence, proposed models are not flexible enough, since for each delegation requirement several components are added. This will complicate the administration task given the fact that there are many roles and



Rule name	Role	Activity	View	Context
p8	EC	approuver_ec	vDemande_achat	ctx_demande_bibliotheque
p10	rService_financier	approuver_sf	vDemande_achat	ctx_livre_onereux
p9	Doyen	approuver_doyen	vDemande_achat	ctx_demande_bibliotheque

Delegate	Rule name	Role	Activity	View	Context
☐	ap5	admin	manage	view_hierarchy...	default_context
☑	root_license	admin	manage	license_view	default_context
☐	ap4	admin	manage	view_view	default_context

Delegation options
Grantee: Marcel
☐ Transfert license
☑ With grant option
Grantee: Erwan
Grant context: default_context
Steps: 1

Fig. 10 The delegation MotOrBAC GUI

permissions to manage; moreover, this requires the specification of a separate security policy dedicated to delegation components.

For instance, in the RBDM0 model proposed by [4], authors extend the RBAC_0 model to define role-based delegation. They define a relationship $can\_delegate \subseteq R\text{x}R$ to control role delegation and add new components such as: $Users\_O$ and $Users\_D$ to differentiate between original and delegated members, UAO and UAD to specify original member assignment and delegate member assignment relationships. $(x, y) \in can\_delegate$ means that a grantor who is an original member of role x can delegate role x to any grantee who is an original member of role y. The right to revoke is defined implicitly: any user empowered in role r is allowed to revoke r.

They also propose some extensions to RBDM0 to address more delegation characteristics. This requires additional components. For instance, to model partial role delegation they add new types of permissions: delegable (PD) and non delegable permissions (PN) and divide the PA relationship (permission-role assignment) into PDA and PNA (Delegable and Non-Delegable Permission Assignment). Similarly, to deal with the two step delegation, they divide the UA relationship (user-role assignment) and $Users$ set into 3 layers. RBDM1 [5] is an extension of RBDM0 that is proposed to deal with the hierarchical roles. The relationship $can\_revoke \subseteq R\text{x}R$ is defined to control revocation.

More recently, authors extend the RBDM model to deal with agent-based delegation (ABRDM [6]). They add a new type of role AR (Agent Role) and a new relationship UAA (agent member to role assignment). To control delegation they define the relationship $can\_delegate \subseteq R\text{x}AR\text{x}R$. They also extend their model to deal with agent-based delegation in hierarchical roles (ARBDMH) and introduce the notion of prerequisite conditions CR. To control delegation the following relationship is used: $can\_delegate \subseteq AR\text{x}CR\text{x}R$. Note that CR is a set of conditions on the grantee's role.

RDM2000 [32] is also a delegation model based on RBDM0. This model introduces new components to deal with multi-step delegation and constraints, such as DT (Delegation Tree), $Path$ (delegation path), $Depth$ (delegation depth) and the relationship $can\_delegate \subseteq R\text{x}CR\text{x}N$, where N is the max delegation depth. The delegation relation $DLGT$ is also added to address the relationship between different components involved in a delegation. This relationship is divided into $OLDGT$ and $DDLGT$ (Original and Delegated user delegation). To control Grant-Independent and Grant-Dependent revocation two relationships are added: $can\_revokeGI \subseteq R$ and $can\_revokeGD \subseteq R$.

Other works [21,2] have proposed to extend the RDM 2000 model. The first model adds several new components to deal with role-to-role delegation and partial delegation, and the second introduces a constraint specification to deal with the separation of duty.

PBDM [33] is another delegation model based on RBAC. This model adds new types of roles and permissions to address permission level delegation requirements. In PBDM0 roles are partitioned into regular roles (*RR*) and delegation roles (*DTR*). This partition induces a parallel partition of the two RBAC components: *UA* (user-role assignment) and *PA* (permission-role assignment). Like in RBDM, *UA* is separated into *UAR* (user-regular role assignment) and UAD (user-delegation role assignment). *PA* is similarly separated into *PAR* (permission-regular role assignment) and *PAD* (permission-delegation role assignment). Although this model supports permission delegation, one must use roles to perform a partial delegation: the grantor must create a temporary delegation role (DTR), assigns permission to this role and finally assigns the grantee to this role. To control delegation the can_delegate relationship is defined as follows: $can\_delegate \subseteq RR\text{x}Pre\_con\text{x}P\_range\text{x}M$, where Pre_con: prerequisite condition, P_range: delegation range and M: maximum delegation depth. The right to revoke is implicitly defined: a delegation role is revoked only by owner.

PBDM1 is an extension to PBDM0 which allows the administrator to restrict the delegation. This model adds new components such as delegatable roles (*DBR*), user-delegatable role assignment (*UAB*) and permission-delegatable role assignment (*PAB*). Only permissions assigned to delegatable roles can be delegated. PBDM2 is another extension which addresses role-to-role delegation. This model divides roles into four layers and adds new types of permissions.

In [23], authors propose to extend the PBDM model to deal with the separation of duty and roles inheritance. This model defines six different layers of roles and this partition induces a parallel partition of PA which is separated into seven partitions. It also defines a set of components to describe the static and the dynamic separation of duty constraints (*mutex_roles_set*, *mutex_permissions_set*, *dynamic_ mutex_roles_ set* and *dynamic_mutex_permissions_set*).

The ABDM model [31] is an attribute-based delegation model, which extends PBDM model to address delegation constraints. This model defines two types of constraints: *DAE* (Delegation Attribute Expression) and *CR* (prerequisite conditions). DAE indicates the grantee's qualifications and abilities, and CR indicates prerequisite conditions. There are two types of delegation: decided-delegatees (specified by the grantor) and undecided-delegatees (a qualified grantee candidates generated automatically by the system). The relationship UA is divided into URA (user to regular role assignment), UDA (decided-delegatee to temporary role assignment) and UEA (undecided-delegatee to temporary role assignment). To restrict decided delegation the following relationship is defined: $can\_delegateD \subseteq R\text{x}CR\text{x}DAE\text{x}TDR$. Similarly the relationship $can\_delegateU$ is used to restrict undecided delegation.

The ABDM model is also extended to ABDM$_x$ to deal with temporal delegation. For this purpose, several components are added. For instance, the set of permissions is divided into two types: MP (Monotonous Permission) that can be temporarily or permanently delegated to a qualified user and NMP (Non-Monotonous Permission) that can only be temporarily delegated to a low level grantee. Moreover, to control delegation five relationships are defined: $can\_delegateM$, $can\_delegateTN$, $can\_delegatePN$, $can\_delegateMU$ and $can\_delegatePU$.

URDM [24] is a delegation model based on ABDM and RDM2000. It deals with multiple delegation and introduces new types of constraints. A new delegation relationship called Simultaneous Delegation Relation (SDR) is proposed in this model. This relationship contains two sets of components: original user assignments UAO, and delegated user assignments UAD. Five types of constraints are addressed in the relationship $can\_delegate \subseteq R\text{x}PC\text{x}DW\text{x}DD\text{x}DAE\text{x}DT$, where PC: delegation prerequisite conditions, DAE: delegation attribute expressions, DW: delegation width, DD: delegation depth (only single delegation is supported, so $DD = 1$) and DT: delegation time. PC and DAE are used to identify which grantee satisfies the delegation requirements. The DW constraint is used to deal with multiple delegation, DD to deal with multi-step delegation and DT is the constraint that refers to temporary delegation limited by time. The revocation is controlled by the following relation: $can\_revoke \subseteq R\text{x}CU\text{x}RAE\text{x}DT$, where CU is a set of current users and RAE is a set of attribute expressions for revocation requirements. This model defines more delegation and revocation constraints than the others. But each constraint is defined separately using a specific delegation relation, such as DW, DD and DT. By contrast, in our model we deal with all these constraints using contexts. Hence, to express new constraints we do not specify new relationships but simply new contexts.

Another delegation model is proposed by Crampton and Khambhammettu [11]. It extends RBAC by adding new components to specify delegation characteristics like transfer. It introduces two relationships: $can\_delegate \subseteq R\text{x}R$ and $can\_receive \subseteq R\text{x}C$, where $C \subseteq R$, to specify whether the grantor is authorized to delegate a role and whether the grantee is authorized to receive a role delegation, respectively. The two relationships $can\_delegatep \subseteq R\text{x}P$ and $can\_receivep \subseteq P\text{x}C$ are defined to specify whether the grantor and the grantee are authorized to delegate and receive a permission delegation, respectively. Also, it defines specific actions to perform delegation and transfer, like $grantR_1(u, v, r)$, $grantP_1(u, v, p)$, $xferP_1(u, v, p)$, $xferRstrong(u, v, r)$, $xferRstatic(u, v, r)$ and $xferRdynamic(u, v, r)$, where u is the grantor, v is the grantee, r is the delegated role and p is the delegated permission. $xferRstrong$ and $xferRstatic$ are used, respectively, to transfer a role with and without sub-role inheritance. In our model, we only support role transfer (and delegation) with sub-role inheritance and we consider that if the grantor wants to transfer (or delegate) a role without sub-role inheritance, he/she may transfer (or delegate) all the role permissions instead of the role. This model also uses a delegation history relationship (DH) to record all delegations that have been made and the two relationships *tempUA* and *tempPA* to record temporary user-role and user-permission delegations. Note that in our model we do not need to use such relationships and operations to perform and control delegation, transfer and revocation. The delegation is performed just like the administration, i.e using objects having specific semantics.

In [25], authors propose to extend the RBAC model to deal with delegation in pervasive computing, where dynamic access control is required. In this work only temporal and spatial conditions are supported and the right to delegate is implicitly defined, this means that users are allowed to delegate their own roles and permissions. Moreover and again, like in the other models, specific relationships are defined for each level of delegation. For instance, predicates $Delegate U2U\_P_u(u, v, Perm)$, $DelegateU2U\_P_t(u, v, Perm, d)$, $DelegateU2U\_P_l(u, v, Perm, l)$ and $DelegateU2U\_P_{tl}(u, v, Perm, d, l)$ are used to allow user u to delegate permission $Perm$ to user v without restrictions, for the duration d, in the location l and for the duration d in the location l, respectively. These relationships are defined for user-user permission delegation. Other relationships are also defined for the user–user role delegation, role–role permission delegation and finally role–role delegation.

Compared to these works, our model is more flexible, simpler to manage and more comprehensive. There is no strict distinction between administration and delegation, since there is no specific roles, permissions or relationships for the delegation tasks like in other models. Our delegation model is completely homogeneous with the administration model, hence, we deal with the administration and delegation in a homogeneous unified framework. This is very useful to simplify the administrator task. Indeed, managing delegation do not necessitate the specification of a separate security policy. This is made possible thanks to our extended RBAC model which offers facilities to deal with delegation requirements without requiring additional components. Namely, our model is based on multi-granular and contextual licences, which provides means to define many delegation and revocation requirements as we have shown in this paper.

Moreover, the delegation process in our model is more flexible than in other proposed models. For instance, to delegate a permission the grantor (if he/she is authorized to do so) must simply insert an object with the corresponding delegation attributes in the view *licence_delegation*, and he/she can specify complex conditions to restrict the delegated permissions using contexts. Whereas in other models such as PBDM, ABDM and URDM, the grantor must create a Delegation Role (DTR), assign the delegated permission(s) to this role and finally assign this role to the grantee and the only conditions that he/she can specify concerns the timeout or the delegation depth in the case of multi-step delegation. As we have mentioned previously, this approach is complex to manage since during the delegation process new roles are created and this requires the specification of a separate security policy to deal with delegation and revocation authorizations, hierarchical relationships and global constraints like in [20,23].

To sum up, we give in Table 1 a comparison between our model and some of the aforementioned models. We show how to manage the right to delegate, transfer and revoke permissions and roles. In our model, these rights are managed using "classic" permissions and prohibitions, but other models use specific relationships.

We also show how the administrator can restrict the right to delegate using constraints. In our mode,l these constraints are defined using contexts. Hence, we may define various conditions concerning the delegated privileges, grantor/grantee's characteristics (e.g. roles, attributes), actions (e.g. previous delegation), locations and also concerning the system attributes such as time and circumstance. These conditions can also be specified by users to restrict the scope of their delegated rights and to control the propagation of the delegation. RBAC-based delegation models support only some of these conditions, and they use specific relationships for each constraint level (e.g. time, role, delegation depth). This is not a well-suited approach since the specification of new features is a fastidious task and requires the design of new models. Moreover, models such as DAC do not support such conditions since the grant command does not restrict the usage of information once delegated. Thus, it does not provide accurate information on the permissions propagation, and there is a risk of losing control over delegation [27].

Finally, we present the different delegation features supported by our model. We show that our model is more flexible and comprehensive than other models. It supports a large spectrum of different delegation, transfer and revocation features and can be easily extended to include other features. One may notice that temporary role delegation and transfer are not considered in our model. In fact, in our approach, a context is actually not attached to the role but to the security rules. We argue that the notion of contextual role that was suggested in several models generally corresponds to artificial roles and sometimes to misleading roles (see [12] for a discussion).

8 Conclusion and future work

In this paper, we have proposed a new delegation approach for role-based access control. We have showed that it is possible to specify delegation requirements using a formalism based on the OrBAC model. This model is self-administrated and provides facilities, such as multi-granular licence, contextual licence and use of views, which give means to specify delegation characteristics just like the administration. Therefore our approach is more flexible, simpler to manage and more comprehensive than previous works based on the RBAC model.

Our model supports several delegation features such as totality, permanence, multiple and multi-step delegation,

Table 1 Managing delegation

Characteristic	Our model	RBDM	RDM2000	PBDM	ABDM	URDM	Crampton et al.
Role Delegation (RD)	Permission/Prohibition	can_delegate	can_delegate	can_delegate	can_delegateD can_delegateU	can_delegate	can_delegate can_receive
Constraints	*Contexts*	*Role*	*Role* +*Deleg. depth*	*Role* +*Deleg. depth*	*Role* + *Attributes*	*Role*+ *Time Attributes*+ *Deleg. depth*+ *Deleg. width*	*Role*
Permission Delegation (PD)	Permission/Prohibition	x	x	can_delegate	can_delegateD can_delegateU	can_delegate	can_delegatep can_receivep
Constraints	*Contexts*			*Role* + *Permission*+ *Deleg. depth*	*Role* + *Attributes*	*Role*+ *Time Attributes*+ *Deleg. depth*+ *Deleg. width*	*Role* + *Permission*
Role Transfer (RT_r)	Permission/Prohibition	x	x	x	x	x	can_delegate/can_receive
Constraints	*Contexts*						*Role*
Permission Transfer (PT_r)	Permission/Prohibition	x	x	x	x	x	can_delegatep can_receivep
Constraints	*Contexts*						*Role* + *Permission*
Role Revocation (RR_v)	Permission/Prohibition	can_revoke	can_revokeGD can_revokeGI	implicit	implicit	can_revoke	implicit
Constraints	*Contexts*	*Role*	*Role*	x	x	*Role* + *Time* + *Attributes*	x
Permission Revocation (PR_v)	Permission/Prohibition	x	x	implicit	implicit	can_revoke	implicit
Constraints	*Contexts*			x	x	*Role* + *Time* + *Attributes*	x
Role Transfer Revocation (RTR_v)	Permission/Prohibition	x	x	x	x	x	implicit
Constraints	*Contexts*						x
Perm.Transfer Revocation (PTR_v)	Permission/Prohibition	x	x	x	x	x	implicit
Constraints	*Contexts*						x
Supported Features	Simple/Multiple $RD/RT_r/PD/PT_r$	Simple RD	Simple RD	Simple RD/PD	Simple RD/PD	Simple/Multiple RD/PD	Simple$RD/PD/RT_r/PT_r$
	Single/Multi-step $RD/RT_r/PD/PT_r$	Single/Two-step RD	Single/Multi-step RD	Single/Multi-step RD/PD	Single-step RD/PD	Single-step RD/PD	Single-step$RD/PD/RT_r/PT_r$
	Self/Agent-acted $RD/RT_r/PD/PT_r$	Self-acted RD	Self-acted RD	Self-acted RD/PD	Self-acted RD/PD	Self-acted RD/PD	Self-acted $RD/PD/RT_r/PT_r$
	Temporary PD/PT_r	Temporary RD	Temporary RD	x	Temporary RD/PD	x	Temporary $RD/PD/RT_r/PT_r$
	GD/GI $RR_v/PR_v/RTR_v/PTR_v$	GD/GI RR_v	GD/GI RR_v	GD RR_v/PR_v	GD RR_v/PR_v	GD RR_v/PR_v	GD $RR_v/PR_v/RTR_v/PTR_v$

transfer and agent-acted delegation. These features are managed in a simple manner using contexts. We gave some examples to show how we deal with these aspects and how we can enrich our model to support other delegation requirements. We have also showed that we can deal with several revocation schemes such as dominance, propagation and dependency. More details about revocation mechanism are given in [9].

The OrBAC model being self-administrated, specifying the administration and the delegation policy is done using the same concepts involved in the specification of the security policy, which does not require the administrator to learn a separate administration and delegation models. Hence, we have proposed a single tool called MotOrBAC to manage all these aspects. Although this tool is based on a unique API and uses the same formalism to manage administration and delegation, there is no confusion between these two concepts. Administrative privileges are managed using administrative views, whereas delegation privileges are managed using delegation views. Therefore, in the MotOrBAC tool there is two different GUI to distinguish between the design time when administrators specify and manage the security policy and the run time when users delegate and revoke their rights.

Many more features still remain to be studied in our model. First, a potential work includes the application of our delegation model to specific access control areas, such as workflow systems. Indeed, in [3] authors have proposed a Workflow Management Systems (WFMS) associated with a security policy expressed using the OrBAC formalism. They have shown that OrBAC provides useful notions, such as organization and contexts, and therefore is well suited to manage workflow systems requirements. Therefore, it is possible to apply our approach to WFMS to manage delegation in such systems.

Second, we have only investigated the issue of delegation within a single organization. It is possible to explore delegation requirements when organizations want to interoperate. This means that to allow users to delegate their privileges or responsibilities to external users (i.e. from other organizations) with respect to the interoperability policy. Our work can be extended by the O2O concept (for Organization to Organization) defined in [14], to deal with such requirements.

Delegation of obligations is another issue to address. We have proposed a first model in [10] to manage obligation delegation requirements such as the bilateral agreement delegation and the delegation with responsibility. We plan to further develop this point in future work.

Acknowledgments This work is partially supported by the ANR project Polux. For this work, Meriam Ben-Ghorbel-Talbi is funded by the IFC, the French Institute for Cooperation in Tunisia.

References

1. Abou-El-Kalam, A., Benferhat, S., Miège, A., Baida, R.E., Cuppens, F., Saurel, C., Balbiani, P., Deswarte, Y., Trouessin, G.: Organization based access control. In: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003). IEEE Computer Society (2003)
2. Ahn, G.J., Mohan, B., Hong, S.P.: Towards secure information sharing using role-based delegation. J. Netw. Comput. Appl. **30**(1), 42–59 (2007)
3. Ayed, S., Cuppens-Boulahia, N., Cuppens, F.: Deploying security policy in intra and inter workflows management systems. In: Proceedings of 3rd International Conference on Availability, Reliability and Security (ARES 2009). IEEE Computer Society, Fukuoka (2009)
4. Barka, E., Sandhu, R.: A role-based delegation model and some extensions. In: Proceedings of the 23rd National Information Systems Security Conference (NISSC 2000). Baltimore, MD (2000)
5. Barka, E., Sandhu, R.: Role-based delegation model/hierarchical roles (RBDM1). In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004). Tucson, Arizona (2004)
6. Barka, E., Sandhu, R.: Framework for agent-based role delegation. In: Proceedings of the IEEE International Conference on Communications (ICC 2007). (2007)
7. Ben-Ghorbel-Talbi, M.: Decentralized administration of security policies. Ph.D. Thesis, TELECOM Bretagne-Sup'Com Tunis (2009)
8. Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., Bouhoula, A.: Managing delegation in access control models. In: Proceedings of the 15th International Conference on Advanced Computing and Communications (ADCOM 2007), pp. 744–751. IEEE Computer Society, Guwahati, Inde (2007)
9. Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., Bouhoula, A.: Revocations schemes for delegation licences. In: Proceedings of the 10th International Conference on Information and Communications Security (ICICS 2008). Springer, Birmingham (2008)
10. Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., Bouhoula, A.: An extended role-based access control model for delegating obligations. In: Proceedings of the 6th International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2009). LNCS Springer, Linz, Austria (2009)
11. Crampton, J., Khambhammettu, H.: Delegation in role-based access control. Int. J. Inf. Secur. (2008)
12. Cuppens, F., Cuppens-Boulahia., N.: Modeling contextual security policies. Int. J. Inf. Secur. (2008)
13. Cuppens, F., Cuppens-Boulahia, N., Ben-Ghorbel, M.: High level conflict management strategies in advanced access control models. Electron. Notes Theor. Comput. Sci. (ENTCS) **186**, 3–26 (2007)
14. Cuppens, F., Cuppens-Boulahia, N., Coma, C.: O2O: Virtual private organizations to manage security policy interoperability. In: Proceedings of the 2nd International Conference on Information Systems Security (ICISS 2006), India (2006)
15. Cuppens, F., Cuppens-Boulahia, N., Coma, C.: Multi-granular licences to decentralize security administration. In: Proceedings of the First International Workshop on Reliability, Availability and Security (SSS/WRAS 2007). Paris, France (2007)
16. Cuppens, F., Cuppens-Boulahia, N., Miège, A.: Inheritance hierarchies in the Or-BAC model and application in a network environment. In: Proceedings of the 3rd Workshop on Foundations of Computer Security (FCS04). Turku, Finland (2004)
17. Cuppens, F., Miège, A.: Administration model for Or-BAC. Int. J. Comput. Syst. Sci. Eng. (CSSE) **19**(3) (2004)

18. Griffiths, P.P., Wade, B.W.: An authorization mechanism for a relational database system. ACM Trans. Database Syst. **1**(3) (1976)
19. Jena: A Semantic Web Framework for Java. http://jena.sourceforge.net/
20. Kong, G., Li, J.: Research on RBAC-based separation of duty constraints. J. Inf. Comput. Sci. **2**(3), 235–240 (2007)
21. Lee, Y., Park, J., Lee, H., Noh, B.: A rule-based delegation model for restricted permission inheritance RBAC. In: Proceedings of the 2nd International Conference (ACNS 2004). Yellow Mountain (2004)
22. Motorbac: http://motorbac.sourceforge.net/
23. Park, D.G., Lee, Y.R.: A flexible role-based delegation model using characteristics of permissions. In: Proceedings of the 16th International Conference on Database and Expert Systems Applications (DEXA 2005). Copenhagen, Denmark (2005)
24. Qiu, W., Adams, C.: Exploring user-to-role delegation in role-based access control. In: Proceedings of the 8th World Congress on the Management of eBusiness (WCMeB 2007). IEEE Computer Society, Toronto, ON (2007)
25. Ray, I., Toahchoodee, M.: A spatio-temporal access control model supporting delegation for pervasive computing applications. In: Proceedings of the 5th International Conference on Trust, Privacy & Security in Digital Business (TrustBus'08). LNCS Springer, Turin (2008)
26. Sandhu, R., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Comput. **29**(2), 38–47 (1996)
27. Sandhu, R.S., Samarati, P.: Access control: principles and practice. lEEE Commun. Mag. (1994)
28. The Motorbac Tool: http://motorbac.sourceforge.net/
29. Toman, D.: Memoing evaluation for constraint extensions of datalog. Constraints **2**(3/4), 337–359 (1997)
30. Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies. W.H. Freeman & Co, New York, NY (1990)
31. Ye, C., Wu, Z., Fu, Y.: An attribute-based delegation model and its extension. J. Res. Pract. Inf. Technol. **38**(1) (2006)
32. Zhang, L., Ahn, G.J., Chu, B.T.: A rule-based framework for role-based delegation and revocation. ACM Trans. Inf. Syst. Secur. (TISSEC) **6**, 404–441 (2003)
33. Zhang, X., Oh, S., Sandhu, R.: Pbdm: a flexible delegation model in RBAC. In: Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003). ACM Press, Como (2003)

Author biographies

Meriam Ben-Ghorbel-Talbi is currently a postdoc at the TELECOM Bretagne. She received her Ph.D in computer science, in 2009, co-supervised with TELECOM Bretagne and Sup'Com the Engineering School of Telecommunications in Tunisia. She holds a Master degree, in 2005, in computer science from ENSI the National School of Computer Science in Tunisia and an Engineering degree, in 2003, in network and computer science from INSAT the National Institute of Applied Sciences and Technologies in Tunisia. Her research interests include access control policies and intrusion detection systems.

Frédéric Cuppens is a full professor at TELECOM Bretagne. He holds an engineering degree in computer science, a Ph.D and an HDR. He has been working for more than 15 years on various topics of computer security including definition of formal models of security policies, access control to network and information systems, intrusion detection and formal techniques to refine security policies and prove security properties. He has published more than 100 technical papers in refereed journals and conference proceedings. He served on several international conference programme committees and was the Programme Committee Chair of ESORICS 2000, IFIP SEC 2004 and SAR2006.

Nora Cuppens-Boulahia is a teacher/researcher at TELECOM Bretagne. She holds an engineering degree in computer science and a Ph.D from ENSAE and an HDR. Her research interest includes formalization of security policies and security properties, cryptographic protocol analysis and formal validation of security properties. She has published more than 50 technical papers in refereed journals and conference proceedings. She has been member of several international program committees in information security system domain and the Programme Committee Chair of Setop 2008 and SAR-SSI 2008. She is member of the editorial board of "Computer & Security" journal, she is the French representative of IFIP TC11 "Information Security," and she is co-responsible of the information system security pole of SEE-TIC.

Adel Bouhoula received in 1990 the Diploma degree in computer engineering with Distinction from the University of Tunis (Tunisia). In 1991, he received a Masters degree, in 1994 a Ph.D degree with Distinction and in 1998 the Habilitation degree all in computer science from Henri Poincare University in Nancy (France). Adel Bouhoula is currently a full Professor at the Sup'Com Engineering School of Telecommunications in Tunisia. He is also the Head of the "Digital Security" Research Unit and the President of the Tunisian Association of Digital Security. His research interests include Automated Reasoning, Network Security, Cryptography and Validation of cryptographic protocols.