An Extended Authorization Model for Relational Databases

Elisa Bertino, *Member*, *IEEE*, Pierangela Samarati, and Sushil Jajodia, *Senior Member*, *IEEE*

Abstract—We propose two extensions to the authorization model for relational databases defined originally by Griffiths and Wade. The first extension concerns a new type of revoke operation, called *noncascading* revoke operation. The original model contains a single, *cascading* revoke operation, meaning that when a privilege is revoked from a user, a recursive revocation takes place that deletes all authorizations granted by this user that do not have other supporting authorizations. The new type of revocation avoids the recursive revocation of authorizations. The second extension concerns *negative* authorization which permits specification of explicit denial for a user to access an object under a particular mode. We also address the management of views and groups with respect to the proposed extensions.

Index Terms—Database systems, relational database, access control, authorization, security, protection, privacy, revocation of authorizations.

1 Introduction

Authorization is an important functionality that any multiuser database management system (DBMS) must provide. When several users and applications need to share data of different degrees of sensitivity, it is important that the DBMS provides capabilities to define and enforce access control policies. Therefore, the problem of defining suitable authorization models and the proper techniques to support them has been a relevant aspect in the development of DBMS. An important milestone in the history of authorization is the model defined by Griffiths and Wade [14], in the framework of the relational DBMS System R [1]. This model introduced the notion of decentralized administration of authorizations, based on the principle that the creator of an object is the administrator of the authorizations on the object and is allowed to delegate this function to other users by giving them authorizations with the grant option. Any user who has the authorization for a privilege on a table with the grant option can administer the privilege on the table (i.e., grant to or revoke from other users authorizations on the table).

Another important contribution of the System R authorization model is that it permits content-dependent authorizations, in addition to content-independent authorizations. The approach proposed by Griffiths and Wade is based on views. Since a view is defined in terms of a query, it is possible to define views containing predicates or statistical summaries computed over table columns. Therefore, if an authorization is granted to a user on a view, rather than on a table, the user only sees the data that are filtered through the view, rather than the entire table.

The System R authorization model has been extended by Wilms and Lindsay [25] with functionalities for group management to allow authorizations to be granted to a group of users, rather than to a single user. This functionality is crucial for many organizations where groups of users may perform similar tasks or cooperate on a task and, therefore, need the same authorizations. Wilms and Lindsay also extend the authorization mechanism to the framework of the distributed DBMS System R*. The authorization model of System R* has been further extended by Bertino and Haas to deal with distributed views [6].

In this paper, we present two major extensions to the System R authorization model. We base our work on the System R model for the following reasons. First, this model is the basis on which authorization models, used in commercial relational DBMS, have been developed. Note that commercial systems typically augment the System R model by introducing new authorization types (such as the reference authorization needed to define referential integrity constraints) to take into account the new functionalities that have been added to the data model. Our extensions can be orthogonally applied to the current authorization models as well. Second, a formal foundation has been established for the System R model [12]. One of the goals of our work is to investigate how this formal model can be extended to account for new features of advanced authorization models.

The first of our extensions introduces a new type of revoke operation. In the original proposal by Griffiths and Wade, if an authorization is revoked from a user, a recursive revoke is applied whenever the authorization being removed has the *grant* option specified. The recursive revoke removes authorizations that have been granted by the user from whom the authorization is being revoked and do not have other supporting authorizations. We call this approach *cascading revoke*. A problem with this approach is

- *E. Bertino and P. Samarati are with Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy. E-mail: {bertino, samarati}@dsi.unimi.it.*
- *S. Jajodia is with the Center for Secure Information Systems and the Department of Information and Software Systems Engineering, George Mason University, 4400 University Dr., Fairfax, VA 22030. E-mail: jajodia@gmu.edu.*

Manuscript received Mar. 29, 1994; revised Oct. 31, 1994.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96097.

that it can be very disruptive sometimes [3]. Indeed, in many organizations, the authorizations users possess are related to their particular tasks or functions within the organization. Suppose there is a change in the task or function of a user (say, because of a job promotion). This change may imply a change in the responsibilities of the user and therefore in his privileges. New authorizations will be granted to the user and some of his previous authorizations will be revoked. Applying a recursive revocation will result in the undesirable effect of deleting all authorizations the revokee granted and, recursively, all the authorizations granted through them, which then will need to be reissued. Moreover, all application programs depending on the revoked authorizations will be invalidated [8]. Many other examples can be found where the effect of recursively deleting the authorizations upon a revoke is not wanted. We propose a new type of revoke operation, called *noncascading revoke*, which differs from the cascading revoke in that no recursive revocation of authorizations is performed. We believe that both types of revoke should be offered by a DBMS since a choice would then result in greater flexibility. A similar extension has been proposed in a recent draft of the SQL standard [18]; however, the SQL draft does not address the problems related to its application.

The second extension concerns negative authorization. Most DBMS use the *closed world* policy. Under this policy, the lack of an authorization is interpreted as a negative authorization. Therefore, whenever a user tries to access an object, if a *positive* authorization (i.e., an authorization permitting the access) is not found in the system catalogs, the user is denied the access. This approach has a major problem in that the lack of a given authorization for a given user does not prevent this user from receiving this authorization later on. Suppose that a user *x* should not be given access to an object *o*. In situations where authorization administration is decentralized (like in System R), a user possessing the right to administer object *o*, can, perhaps by mistake, grant *x* the authorization on that object. Since several users may have the right to administer the same object, it is not always possible to enforce with certainty the constraint that a user cannot access a particular object. We, therefore, propose an explicit *negative* authorization as an approach for handling this type of constraint. An explicit negative authorization expresses a *denial* for a user to access an object. In our approach negative authorizations are stronger than positive authorizations. That is, whenever a user has both a positive and a negative authorization on the same object, the user is prevented from accessing the object.

Negative authorizations in our model are handled as blocking authorizations. Whenever a user receives a negative authorization, his positive authorizations become blocked. A blocked authorization is not removed from the authorization catalogs. Rather, it is preserved and marked as not usable. If later on, the negative authorization is revoked, the blocked positive authorizations are unblocked and the user can once again use the privileges stated by those positive authorizations. This approach has a number of advantages. First of all, if a negative authorization is given by mistake to a user, it is always possible to recover from the error by simply revoking the negative authorization, since the positive authorizations related to the same privilege are preserved in the authorization catalogs. Second, it is possible to temporarily suspend a privilege from a user by granting him a negative authorization and revoking it later on. In a system without negative authorization, the privilege would have to be revoked and granted again later on. However, a revoke operation has the disruptive effect that all authorizations granted on the basis of the revoked privilege are in turn revoked and, moreover, all views requiring the revoked privilege are deleted. To our knowledge, no current authorization mechanism allows the possibility of temporary suspension of authorizations, as we support in our model through negative authorization.

Negative authorizations are also attractive since they allow exceptions to be specified. Without an exception mechanism, certain requirements can only be expressed by greatly increasing the number of groups or authorizations, making the management of authorizations more difficult. To illustrate, suppose we wish to grant an authorization to all members of a group, except to one specific member *m*. In the absence of negative authorizations, we would have to express the above requirement by specifying a positive authorization for each member of the group except *m*. If negative authorizations are considered, the same requirement can be expressed by granting a positive authorization to the group and a negative authorization to member *m*. Note that the exception handling is included among the requirements for high assurance systems [10], and the need for negative authorizations has been recognized by several researchers ([11], [19], [20], [21], [23]).

The original contributions of our work can be summarized as follows:

1) The formal definition of an authorization model that includes two types of revoke operations (cascading and noncascading) and negative authorization.
2) A detailed investigation of the effects of adding these features on views and groups.

In this paper, we only deal with discretionary access control since the focus of our research is how to extend the authorization facilities provided by commercial relational DBMS (RDBMS) and a large majority of commercial RDBMS only provide discretionary access control. Mandatory access control models have been proposed as an alternative to cope against sophisticated attacks, such as Trojan Horses or covert channels, that discretionary access control mechanisms are unable to prevent [10]. Recent multilevel RDBMS (like Trusted Oracle [17] and Secure Informix [15]) provide mandatory access control coupled with discretionary access control. Therefore, the new features provided by our model could be orthogonally incorporated into such systems as well.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces the basic authorization model that is used and extended in subsequent sections. The model is extended to include cascading and noncascading revoke operations in Section 4, and negative authorization in Section 5. The model is further extended to incorporate views and groups in Sections 6 and 7, respectively. Section 8 concludes the paper.

2 Related Work

Early authorization models only allowed the specification of positive authorizations. More recent authorization models, in the context of operating systems and database systems, allow the specification of negative authorizations stating accesses to be denied. As for operating systems, an authorization model supporting positive as well as negative authorizations has been proposed in the context of the Andrew File System [23]. Subjects of the authorizations can be users and groups of users. A group is a set of groups and users and is associated with an owner who is allowed to add and remove members from the group. A user operates with the union of his authorizations and that of the groups to which he belongs. A user can temporarily disable personal membership to some groups. Authorizations of disabled groups are not available to the user. In case of simultaneous presence of negative and positive authorizations, the model adopts the denials-take-precedence policy, i.e., the considered access is not allowed. The model provides a limited form of authorization administration, where only the owner of a file can grant and revoke authorizations to other subjects. Moreover, since objects of the authorizations can only be files, the problem of authorizations on derived objects does not arise.

As for database systems, a model supporting positive and negative authorizations has been developed at SRI in the context of the SeaView project [20]. Subjects of authorizations can be users as well as groups. Only users can belong to groups, i.e., groups cannot be specified as members of other groups. The SeaView model supports negative authorizations by introducing a special access mode, called "null." Granting the null privilege on an object to a user means denying the user *all* accesses on the object. Thus, if a user is given a null privilege on an object, the user will not be able to access the object, even if he owns an authorization for the access. The administration of privileges is controlled through the access modes "grant" and "give-grant." If a user has the "grant" access mode on a table, he can grant and revoke any access mode (including "null") on the table from other subjects in the system. If a user has the "give-grant" access mode on a table, he can additionally grant and revoke the "grant" and "give-grant" access modes on the table from other subjects in the system. Conflicts among authorizations are solved on the basis of the following policy: 1) authorizations specified for a user take precedence over authorizations specified for the groups to which the user belongs, and 2) the authorization for the null access mode specified for a user (group) takes precedence over any other authorization specified for the same user (group). The SeaView model suffers from several limitations. First, the model does not support negative authorization at the level of a single access mode. Thus, for example, it is not possible to state that a user should be authorized for the select access mode on an object while at the same time denied for the insert access mode on the object. Second, administrative authorizations are referred to all privileges executable on an object and not to single privileges. Thus, it is not possible to give a user the authorization to administer a specific privilege (e.g., select) on a table. Third, only users can belong to groups. Fourth, authorizations specified for groups are considered only when no authorizations are explicitly specified for the user. Hence, if we would like to give a user some authorization in addition to the authorization he has as a member of some groups, we need to respecify the authorizations of the groups for the user himself.

Another model supporting negative authorization has been proposed in the context of object-oriented systems in the framework of the ORION/ITASCA project [21]. Authorizations can be specified only for "roles" (which are groups of users) and not for single users. The model enforces derivation of additional authorizations, called implicit, from the authorizations explicitly specified by the users. In particular, a positive authorization granted to a group implies the derivation of the same authorization for all its subgroups. By contrast, a negative authorization specified for a group implies the derivation of the same negative authorization for the supergroups of this group. Thus negative authorizations propagate up in the group membership graph. Resolution of possible conflicts between positive and negative authorizations is based on the concept of more specific authorization. In evaluating access requests positive authorizations take precedence over negative authorizations: A user can execute an access if at least one of the groups to which he belongs has a (nonoverridden) positive authorization for it, regardless of possible negative authorizations for the access that other groups to which the user belongs may have. The authorization model of ORION suffers from several drawbacks. First, negative authorizations specified for a group do not propagate to its subgroups. This is due to the fact that this model mixes the semantics of user groups with that of user roles [22]. Moreover, the ORION authorization model requires that certain very strict constraints be satisfied by the authorization state. For example, the model requires no conflicts among authorizations and the completeness of the authorization state (i.e., for any possible access, an authorization, either positive or negative, must exist). These requirements complicate authorization management. In particular, the completeness requirement may lead to overloading the specification of authorizations, forcing the users to specify negative authorizations for all accesses which must not be granted. Finally, the ORION model does not address authorization issues with respect to derived objects, like views.

Another model supporting both negative and positive authorizations for protection in object-oriented systems has been proposed by Gal-Oz et al. [11]. Conflicts among positive and negative authorizations are solved using the denials-take-precedence policy. This model does not consider groups of users; authorizations can be specified for single users only. Moreover, the problem of authorization administration is not addressed.

3 Notation and Definitions

In this section we introduce notations and definitions that will be used later on in the paper. Let U be the set of users in the system, S the set of subjects (users and groups), T the

set of objects (i.e., tables), P = {select, insert, delete, update}[1] the set of privileges executable on the objects, and $\mathbb{N}$ the set of natural numbers. An authorization is characterized as follows.

DEFINITION 1. (Authorization) *An authorization a is a six-tuple*

$\langle subject, priv, table, ts, grantor, grant\text{-}opt \rangle$

where:

subject $\in S$ *is the subject (user) to whom the authorization is granted;*

priv $\in P$ *is the privilege;*

table $\in T$ *is the table to which the authorization refers;*

ts $\in \mathbb{N}$ *is the time at which the authorization was granted;*[2]

grantor $\in U$ *is the user who granted the authorization;*

grant-opt $\in$ {*yes, no*} *indicates whether* subject *has the grant option for* priv *on* table.

The above tuple states that user *subject* has been granted *priv* on *table* by user *grantor* at time *ts*. If *grant-opt* = "yes," *subject* has been given the grant option, meaning he is authorized to grant other users *priv* on *table* as well as the grant option on it. The grant option also authorizes the user to revoke *priv* on *table* from other users. Note however that a user can revoke only those authorizations he granted.

For example, tuple $\langle B, select, T, 10, A, yes \rangle$ indicates that user B can select tuples from table T and grant other users authorizations to select tuples from table T, and that this privilege was granted to B by user A at time 10.

In the following, given an authorization a, the notations $subject(a)$, $priv(a)$, $table(a)$, $ts(a)$, $grantor(a)$, $grant\text{-}opt(a)$ denote respectively the subject, the privilege, the table, the time, the grantor, and the grant option in a.

A user creating a table is the owner of the table. As owner, the user is given the authorizations for all the privileges executable on the table with the grant option. We refer to these authorizations as *basic authorizations*. The grantor of the basic authorizations is the system itself, denoted by "*", and the time is the time at which the table was created. Basic authorizations are deleted when the owner drops the table. In our model, only the owner of a table is entitled to drop the table. The model can be easily extended to policies allowing users different from the owner to drop the table by including "drop" in the set of privileges.

DEFINITION 2. (Authorization State (AS)) *An authorization state AS is the set of authorizations present at a given time.*

An authorization state can be represented by a labeled graph as follows. Each user holding some authorization for a privilege on a table is represented by a node labeled with the triple $\langle user, table, privilege \rangle$. An arc between node $n_1 = \langle u_1, t_1, p_1 \rangle$ and node $n_2 = \langle u_2, t_2, p_2 \rangle$, with $t_1 = t_2$ and $p_1 = p_2$, indicates that user u_1 granted an authorization for p_2 on t_2 to user u_2. Every arc is labeled with a triple $\langle a_i, time, grant\text{-}option \rangle$ indicating, respectively, the identifier of the authorization,[3] the time when the authorization was granted, and whether the authorization was granted with the grant option. The symbol "g" associated with the arc indicates that the authorization is with the grant option, whereas, if nothing is specified, the authorization is without the grant option. A special node "*" indicates the system, and arcs leaving from node "*" correspond to basic authorizations. For the sake of simplicity, all examples in the paper will refer to a single privilege (select). Thus, we will omit the privilege in nodes and label each node with a pair $\langle user, table \rangle$. Similarly, we will show only one basic authorization (for the select privilege).

EXAMPLE 1. Suppose authorization state AS consists of the following authorizations:

$a_1 = \langle A, select, T, 10, *, yes \rangle$
$a_2 = \langle B, select, T, 20, A, yes \rangle$
$a_3 = \langle C, select, T, 30, A, yes \rangle$
$a_4 = \langle D, select, T, 40, B, yes \rangle$
$a_5 = \langle E, select, T, 50, D, yes \rangle$
$a_6 = \langle D, select, T, 60, C, yes \rangle$
$a_7 = \langle F, select, T, 70, D, yes \rangle$
$a_8 = \langle G, select, T, 80, E, yes \rangle$.

The graphical representation of AS is illustrated in Fig. 1.

In our authorization model, for every authorization for a privilege on a table, a corresponding authorization must exist for the grantor of the privilege on the table with the grant option. This requirement, is represented by the following relationship among authorizations.

DEFINITION 3. (Supporting Authorization) *Given two authorizations* $a_i, a_j \in AS$, a_i *supports authorization* a_j *(written* $a_i \rightarrow a_j$*) iff* $subject(a_i) = grantor(a_j)$, $priv(a_i) = priv(a_j)$, $table(a_i) = table(a_j)$, $ts(a_i) < ts(a_j)$, $grant\text{-}opt(a_i) =$ "*yes*".

Definition 3 states that authorization a_i supports authorization a_j if the authorizations have the same privilege and table, the subject of a_i is the grantor of a_j and the timestamp of a_i is smaller than the timestamp of a_j. If $a_i \rightarrow a_j$, we say that there exists a *supporting relationship* between a_i and a_j. In the authorization state of Fig. 1, the following supporting relationships hold:

$a_1 \rightarrow a_2$, $a_2 \rightarrow a_4$, $a_4 \rightarrow a_5$, $a_4 \rightarrow a_7$, $a_5 \rightarrow a_8$, $a_1 \rightarrow a_3$, $a_3 \rightarrow a_6$, $a_6 \rightarrow a_7$.

On the basis of the supporting relationship, we introduce the following definition.

DEFINITION 4. (Authorization Chain) *Given an authorization state AS, an authorization chain is a sequence* $\langle a_1, a_2, \ldots, a_n \rangle$ $(n \geq 1)$ *of authorizations such that* $a_1, a_2, \ldots, a_n \in AS$, $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$, and $g(a_1) =$ "*".

In terms of the graph representing the authorization state, a chain corresponds to a path such that all arcs in the path except perhaps the last one are with the grant option, and every authorization has a timestamp greater than the authorization preceding it in the chain.

1. Unlike select, insert, and delete privileges, the update privilege can refer to specific columns inside a table. However, for the sake of simplicity, we consider authorizations as specified on whole tables.

2. A timestamp can be represented by a system maintained counter. Timestamps are necessary to prevent cycles among authorizations.

3. We have associated identifiers with authorizations so that they can be easily referenced.

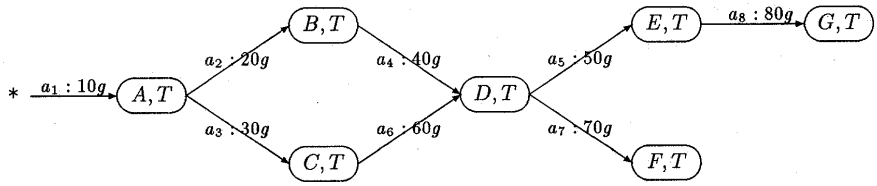


Fig. 1. The graphical representation of authorization state of Example 1.

In the following, $C(AS)$ denotes the set of all authorization chains in authorization state AS. For the authorization state of Fig. 1, the set of chains is as follows:

$$C(AS) = \{\langle a_1\rangle, \langle a_1, a_2\rangle, \langle a_1, a_2, a_4\rangle, \langle a_1, a_2, a_4, a_5\rangle, \langle a_1, a_2, a_4, a_5, a_8\rangle, \langle a_1, a_2, a_4, a_7\rangle, \langle a_1, a_3\rangle, \langle a_1, a_3, a_6\rangle, \langle a_1, a_3, a_6, a_7\rangle\}.$$

Given the definitions above, the requirement that every nonbasic authorization must have a supporting authorization is formalized by the following property.

PROPERTY 1. (Connectivity) An authorization state AS satisfies the Connectivity Property iff $\forall a \in AS : \exists\langle a_1, \ldots, a_n\rangle \in C(AS)$ $(n \geq 1)$ such that $a = a_n$.

DEFINITION 5. *(*Consistency of the Authorization State) *An authorization state AS is consistent iff it satisfies the Connectivity Property.*

The consistency of the authorization state ensures that every authorization in the system has an authorization supporting it. Since an authorization can be revoked only by the user who granted it, the consistency condition ensures that every authorization in the system can be revoked.

4 REVOCATION

4.1 Revocation of Authorizations with Cascade

The semantics of the cascading revocation, as defined by Griffiths and Wade [14], is to produce the authorization state which would have resulted had the revoker never granted the authorizations being revoked. Cascading revocation implies than when a user x revokes from another user y a privilege that x granted to y with the grant option, all authorizations subsequently granted by y, without other supporting authorizations, are recursively revoked.

We formalize the semantics of the recursive revocation by a function *cascade-rvk* (revoke function with cascade) defined next. We use the notation $\langle y, p, t, x\rangle$ to denote a request by user x (revoker) to revoke privilege p on table t from subject y (revokee).

DEFINITION 4. (Revoke Function with Cascade) *Given an authorization state AS and a request for revocation* $\langle y, p, t, x\rangle$, *with* $y \in S$, $p \in P$, $t \in T$, $x \in U$, *cascade-rvk*$(AS, \langle y, p, t, x\rangle)$ *generates a new authorization state* $\overline{AS}$ *defined as follows. Let REV and CREV be the sets of authorizations defined as:*

$$REV = \{a \in AS \mid subject(a) = y, priv(a) = p, table(a) = t, grantor(a) = x\}$$

$$CREV = \{a \in AS \mid \forall\langle a_1, \ldots, a_n, a\rangle \in C(AS), \exists a_i \in REV, 1 \leq i \leq n\}.$$

Then, $\overline{AS} = AS - REV - CREV$.

In the definition, *REV* denotes the set of authorizations being revoked,[4] and *CREV* denotes the set of authorizations which, after deleting the authorizations in *REV*, do not belong to any authorization chain. The authorizations in *CREV* are those authorizations which would not have existed had the revoker never granted the privilege to the revokee.

EXAMPLE 2. Consider the authorization state of Fig. 1 and suppose that user B revokes the select privilege on table T from user D. The authorization to be revoked is the authorization granted by B to D (a_4), i.e., $REV = \{a_4\}$. As a consequence also the authorization granted by D to E (a_5) and by E to G (a_8) must be revoked, i.e., $CREV = \{a_5, a_8\}$. The resulting state, illustrated in Fig. 2, is $\overline{AS} = \{a_1, a_2, a_3, a_6, a_7\}$. Note that the authorization granted to F by D has not been deleted because of the authorization granted to D by C at time 60.

4.2 Revocation of Authorizations without Cascade

The noncascading revocation allows a user to revoke a privilege on a table from another user without entailing automatic revocation of the authorizations for the privilege on the table the latter may have granted. Instead of deleting the authorizations the user may have granted by using the privilege received by the revoker, all these authorizations are restated as if they had been granted by the revoker.

The semantics of the revocation without cascade of privilege p on table t from user y by user x is 1) to restate with x as grantor all the authorizations in AS that y granted by using the authorizations being revoked and 2) to remove from the authorization state all the authorizations which would not have existed if x, instead of granting the privilege to y, had himself granted all the authorizations in AS that y granted by using the authorizations being revoked.

Note that, since y may have received the grant option for the privilege on the table from some other users different from x, not all the authorizations he granted will be deleted or given to x. In particular, x will be considered as grantor only of the authorizations y granted after receiving the grant option on the privilege from x; y will still be considered as grantor of all the authorizations he granted that are supported by other authorizations not granted by x.

The semantics of the revocation is formalized by function *rvk* (revoke function), defined next.

4. Note that *REV* is a set since a single revoke operation may refer to more than one authorization (i.e., x may have granted y more than one authorization for privilege p on table t).

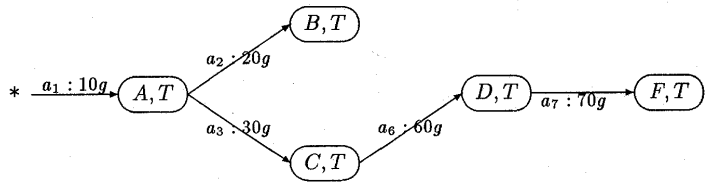


Fig. 2. Authorization state after a_4 is revoked with cascade from Fig. 1.

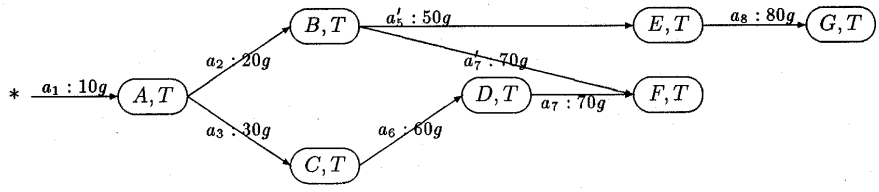


Fig. 3. Authorization state after a_4 is revoked without cascade from Fig. 1.

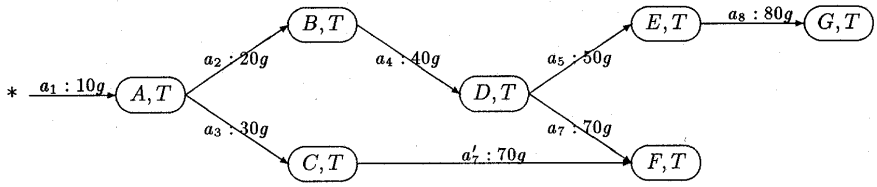


Fig. 4. Authorization state after a_6 is revoked without cascade from Fig. 1.

DEFINITION 7. (Revoke Function) *Given an authorization state AS and a request for revocation* $\langle y, p, t, x\rangle$, *with* $y \in S$, $p \in P$, $t \in T$, $x \in U$, *rvk*$(AS, \langle y, p, t, x\rangle)$ *generates a new authorization state* $\overline{AS}$ defined as follows. Let *REV*, *SUP*, and $\hat{AS}$ *be the sets of authorizations defined as:*

$$REV = \{a \in AS \mid subject(a) = y, priv(a) = p, table(a) = t, grantor(a) = x\}$$

$$SUP = \{\langle subject(a_i), priv(a_i), table(a_i), ts(a_i), x, grant\text{-}opt(a_i)\rangle \mid \exists a_i \in AS, \exists a_j \in REV: subject(a_i) \neq x, subject(a_i) \neq y, a_j \rightarrow a_i\}$$

$$\hat{AS} = AS \cup SUP.$$

Then, $\overline{AS} = cascade\text{-}rvk\ (\hat{AS}, \langle y, p, t, x\rangle)$.

In the above definition, *REV* denotes the set of authorizations being revoked and *SUP* denotes the set of the authorizations which are restated with the revoker as grantor. State $\overline{AS}$ is obtained by adding to *AS* the authorizations that *y* granted by using the authorizations being revoked with *x* as grantor (*SUP*) and then applying cascading revocation.

EXAMPLE 3. Consider the authorization state of Fig. 1 and suppose user *B* revokes the select privilege on table *T* from user *D* without cascade. The authorization to be revoked is the authorization granted by *B* to *D* (a_4), i.e., $REV = \{a_4\}$. The authorizations granted by *D* to *E* (a_5) and *F* (a_7) must be restated with *B* as grantor. Hence, $SUP = \{a_5', a_7'\}$, where $a_5' = \langle E, select, T, 50, B, yes\rangle$ and $a_7' = \langle F, select, T, 70, B, yes\rangle$ derived from a_5 and a_7, respectively. These authorizations are added to *AS* and then function *cascade-rvk* is applied to the resulting state. As a consequence of revoking a_4 also the authorization granted by *D* to *E* (a_5) must be revoked, $CREV = \{a_5\}$. The resulting authorization state, illustrated in Fig. 3, is $AS = \{a_1, a_2, a_3, a_5', a_6, a_7, a_7', a_8\}$.

As another example, consider the authorization state of Fig. 1 and suppose user *C* revokes the select privilege on table *T* from user *D* without cascade. The resulting authorization state is illustrated in Fig. 4. Note that the authorization granted by *D* to *E* has not been specified with *C* as grantor because it was granted before *D* received the privilege from *C*.

Note that in the case of noncascading revocation, a user may become grantor of authorizations he did not grant. This raises the issue of accountability, which can be addressed either by informing the revoker of the authorizations that will be respecified with him as grantor or by keeping track of the user who initially inserted the authorizations by using a log mechanism (see also Section 8).

5 NEGATIVE AUTHORIZATIONS

In this section we extend our model with negative authorizations. In our approach, negative authorizations can only be issued with respect to access privileges, and not to the administration of the privileges themselves. Thus, it is possible to specify that a user cannot select tuples from a table, but it is not possible to specify that a user cannot grant others the authorization to select tuples from the table (i.e., it is not possible to specify the negation for the grant option alone). Note, however, that the negation for a privilege on a table implies the denial of the administration of the privilege itself. For instance, if a user has a negative authorization for the select privilege on table *t*, the user can neither

select tuples from table *t* nor grant or revoke other users the select privilege on table *t*.

To represent negative authorizations, we extend the definition of authorization as follows.

DEFINITION 8. (Authorization) *An authorization a is a seven-tuple*

⟨*subject, priv, priv-type, table, ts, grantor, grant-opt*⟩,

where priv-type ∈ {+, –} *specifies whether the authorization is positive or negative and the other elements have the meaning illustrated in Definition* 1.

Authorizations considered so far have *priv-type* = "+". Authorizations with *priv-type* = "–" have necessarily *grant-opt* = "no," i.e., a negative authorization cannot be specified with the grant option.

If a user has an authorization for a privilege on a table with the grant option, the user can also grant other users negative authorizations for the privilege.[5] Then, a positive authorization with the grant option can support negative authorizations; however, a negative authorization cannot support any authorization (negative or positive).

To represent negative authorizations, we extend our graphical notation by adding the signs of the authorizations to the labels of arcs as follows. A "–" will be associated with the label of an arc if the arc corresponds to a negative authorization, whereas nothing will be indicated if the arc corresponds to a positive authorization.

In the remainder of this section we revisit grant and revoke operations to take into account negative authorizations.

5.1 Grant Operation for Negative Authorizations

5.1.1 Authorizations of the User Receiving Negative Authorization

Negative authorizations introduce the possibility of conflicts. A negative authorization states that a user must be denied a privilege, whereas a positive authorization states that a user must be given a privilege. It should be the case, therefore, that if user *y* has a negative authorization for privilege *p* on table *t*, then *y* should not have at the same time a positive authorization for *p* on t, and vice versa. This situation, although desirable, cannot always be satisfied.

To see this, suppose that a user cannot have at the same time both positive and negative authorizations for a privilege on a table. Suppose that user *x* grants a negative authorization for privilege *p* on table *t* to user *y*, who has some positive authorizations for *p* on *t*. Since *y* cannot have at the same time the positive and the negative authorization, then either all *y*'s positive authorizations should be revoked or the grant request should be refused by the system. Deleting *y*'s positive authorizations is not the correct approach, since it may cause deletion of authorizations granted to *y* by users different from *x*. This is contrary to the assumption that a user can revoke only the authorizations he granted. On the other hand, the approach of rejecting the insertion of the negative authorization would deny *x* the execution of the grant operation for which he has the necessary authorization. Therefore, we allow the insertion of a negative authorization without deleting possible positive authorizations that *y* may have.

The simultaneous presence of positive and negative authorizations does not have to be interpreted as an inconsistency. In our model, negative authorizations override positive authorizations: if a user has a negative authorization: for a privilege on a table, the user will not be able to use any possible positive authorizations for the privilege on the table he may have. In this case, the positive authorizations are said to be *blocked*. This is formalized by the following definition.

DEFINITION 9. (Blocked Authorization) *An authorization* $a_i \in AS$, *with* $grantor(a_i) \neq$ "*", *and with* $priv\text{-}type(a_i) =$ "+", *is blocked for user* $u = subject(a_i)$[6] *iff there exists an authorization* $a_j \in AS$ *such that* $subject(a_i) = subject(a_j)$, $priv(a_i) = priv(a_j)$, $priv\text{-}type(a_j) =$ "–", $table(a_i) = table(a_j)$.

Authorization a_j is called a *blocking* authorization for a_i for user u. We use the notation $a_j \dashv_u a_i$ to indicate that authorization a_j blocks authorization a_i.

Condition "$grantor(a_i) \neq$ *" in the definition expresses the fact that basic authorizations, i.e., authorizations of the owner of a table on the table, can never be blocked.

The time at which an authorization becomes blocked is defined as follows.

DEFINITION 10. (Blocking Time) *Given a blocked authorization* $a_i \in AS$, *the blocking time of* a_i *for user* $u = subject(a_i)$ *is defined as the maximum between the time of* a_i *and the minimum time of the authorizations blocking it. Formally:*

$$bt(a_i, u) = \max(ts(a_i), \min(\{ts(a_k) \mid a_k \dashv_u a_i\})).$$

EXAMPLE 4. Consider the authorization state illustrated in Fig. 2 and suppose that, at time 80, user *B* grants user *D* a negative authorization for the select privilege on table *T* (see Fig. 5). As a result, authorization a_6 granted by *C* to *D* becomes blocked. The blocking time of the authorization $bt(a_6, D)$ is 80.

Authorization a_6 will no longer be blocked in case blocking authorization a_8 is deleted. The deletion of a_8 will occur if either *B* revokes the negation for the select privilege on *T* from user *D* or user *A* revokes the select privilege on *T* from user *B* with cascade.

5.1.2 Authorizations Granted by the User Receiving Negative Authorization

An important issue concerns the authorizations that have been granted by a user who receives a negative authorization. Whenever a user *y* receives a negative authorization for a privilege on a table, all *y*'s authorizations for the privilege on the table become blocked and, therefore, *y* will not be able to revoke authorizations for the privilege on the table he may have granted to others. For example, with reference to the authorization state illustrated in Fig. 5, user *D* will not be able to revoke the select privilege on *T* from user *F* (authorization a_7) because of the authorization a_8. This leads

5. Note that this can be seen as being too permissive since any user with a privilege on a table with the grant option can deny access to all users except the creator of the table. One way to prevent this from happening is to tightly control who can administer negative authorizations on a table by introducing different types of privileges for the administration of negative and positive authorizations. We will not make such a distinction in this paper.

6. This condition is not relevant now; we introduce it to avoid extending some definitions later on.

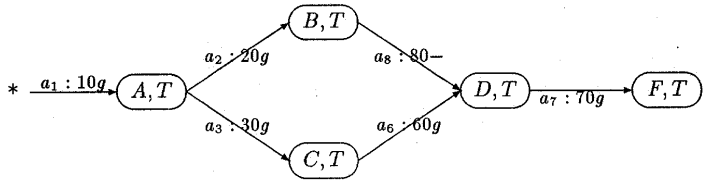


Fig. 5. An authorization state consisting of both positive and negative authorizations.

to the problem of dealing with the authorizations that the user receiving the negative authorization may have granted before receiving the negative authorization. In particular, the question arises whether or not these authorizations should also be recursively blocked. We note that the recursive blocking of these authorizations may not be desirable. Hence, in our model we do not take any particular action over the authorizations granted by user *y* receiving the negation. If the user granting the negative authorization to *y* wishes to additionally block the authorizations for *p* on *t* granted by *y* to other users, he can do so by explicitly granting the negative authorizations to these users.

Therefore, in our model, user *y*, after having received a negative authorization for privilege *p* on table *t*, may still appear as grantor of the authorizations for *p* on *t* granted before receiving the negation. Since *y* can neither grant nor revoke privilege *p* on table *t*, *y* will not be able to revoke these authorizations until his negative authorization will be revoked. Even if these authorizations cannot be revoked by *y*, they can still be removed. They will be revoked upon revocation of the blocked authorizations supporting them, or made revocable upon revocation of the blocking authorizations. For example, although authorization a_7 in Fig. 5, cannot be revoked by *D*, it will be deleted if *A* revokes the select privilege from *C* with cascade or *C* revokes the select privilege from *D* with or without cascade (in this latter case, a new authorization will be specified for *F* with *C* as grantor). It is important to note that, according to Definition 9, the authorizations of the owner of a table cannot become blocked. This ensures that authorizations supported by blocked authorizations can always be revoked.

Although a user *y*, who has received the negative authorization for a privilege on a table can appear as grantor of an authorization for the same privilege to another user, *y* cannot grant any authorization for the privilege on the table after having received the negation. This means that a blocked authorization cannot support authorizations with timestamp bigger than the time at which the authorization became blocked. This requirement is expressed by the following property.

PROPERTY 2. (Blocking Conformity) An authorization state *AS* satisfies the Blocking Conformity Property iff $\forall a_i, a_j \in AS, a_i \rightarrow a_j : \exists a_z, a_z \dashv_u a_i, u = grantor(a_j) \Rightarrow bt(a_i, u) > ts(a_j)$.

We extend the definition of consistency of the authorization state requiring that the Blocking Conformity Property also hold.

DEFINITION 11. (Consistency of the Authorization State) *An authorization state AS is said to be consistent iff it satisfies the Connectivity Property and the Blocking Conformity Property.*

5.2 Revocation of Negative Authorizations

The semantics of the revocation of the negation of privilege *p* on table *t* from user *y* by user *x* is to delete all the negative authorizations for *p* on *t* that *x* granted to *y*. Since no authorization can be supported by a negative authorization, there is no need to propagate the effect of the revocation.

The semantics of the revocation of the negation of a privilege on a table from a user is formalized by function *rvk-ng* (revoke negation function), defined next.

DEFINITION 12. (Revoke Negation Function) *Given authorization state AS and a request for the revocation of a negation* $\langle y, p, t, x\rangle$, *with* $y \in S, p \in P, t \in T, x \in U$, *rvk-ng*$(AS, \langle y, p, t, x\rangle)$ *generates a new authorization state* $\overline{AS}$ *defined as follows:*

$$\overline{AS} = AS - \{a \in AS \mid subject(a) = y, priv(a) = p, priv\text{-}type(a) = \text{"-"}, table(a) = t, grantor(a) = x\}.$$

EXAMPLE 5. Consider the authorization state illustrated in Fig. 5 and suppose that user *B* revokes the negation for the select privilege on table *T* from user *D*. The resulting authorization state is as illustrated in Fig. 2, where authorization a_8 has been deleted.

6 VIEWS

Most RDBMS allow users to define views, expressed in terms of queries against base tables and other views. Views represent a simple and effective mechanism for enforcing content-dependent authorization. For example, suppose user *A* creates table *T* and wants to give user *B* the authorization to select only those tuples of *T* for which the value of attribute a_1 is greater than 1,000. To accomplish this, *A* can simply define a view of the form "select * from *T* where $a_1 > 1{,}000$" on top of *T*, and grant *B* the select authorization on the view.

In the following, we will use the term *table* to refer to either a base table or a view. We will explicitly indicate *base table* or *view* when a distinction is needed.

The relationship between a view and the tables on which the view is defined is formalized by the following definition.

DEFINITION 13. (Referenced Table) *A table t is directly referenced by a view v* (*written* $t \rightarrow v$) *if t is used in the definition of v. A table t is referenced by a view v (written* $t \twoheadrightarrow v$*) if either* $t \rightarrow v$ *or there exist tables* $v_1, \ldots, v_n, n \geq 1$ *such that* $t \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \rightarrow v$.

For any view v, we define a *view derivation graph* as follows. Every table directly or indirectly referenced by a view is represented by a node. There exists an arc directed from table t_i to table t_j iff $t_i \rightarrow t_j$. For example, consider tables T_1, T_2, V_1, V_2, V_3, and V_4, where T_1 and T_2 are base tables, V_1 is defined on top of T_1, V_2 is defined on top of T_1 and T_2, V_3 is defined on top of T_1, and V_4 is defined on top of V_3. The derivation graphs for views V_1, V_2, V_3, and V_4 are illustrated in Fig. 6.

A path between a base table and a view in the derivation graph of the view is called a *derivation path* for the view. In the following, given a view v, $DP(v)$ denotes the set of all the derivation paths for v. For example, with reference to the views in Fig. 6, $DP(V_1) = \{\langle T_1, V_1\rangle\}$, $DP(V_2) = \{\langle T_1, V_2\rangle, \langle T_2, V_2\rangle\}$, $DP(V_3) = \{\langle T_1, V_3\rangle\}$, $DP(V_4) = \{\langle T_1, V_3, V_4\rangle\}$. Base tables have only one derivation path, containing the table itself; thus, for any base table t, $DP(t) = \{\langle t\rangle\}$.

6.1 Authorizations on Views

A user defining a view is its owner and, as such, is entitled to drop the view. However, the view owner may not be authorized to exercise all privileges on the view. The authorizations a view owner has on the view depend on the view semantics (certain operations may not be executable on the view) and on the authorizations the user has on the tables directly referenced by the view.

In the original System R model, in order to be authorized for a privilege on a view, the owner must have authorizations for the privilege on all the tables directly referenced by the view [14].[7] The privileges on the view, however, may be further restricted depending on the view semantics [14]. For example, in most commercial DBMS, updates are not allowed on a view defined as a join of two tables, independently of whether the view owner has the update privileges on these two tables. In the remainder, we will not discuss any further how to determine which update operations can be allowed on a view. We refer the interested reader to the large body of literature dealing with this issue [2], [7], [9], [16]. Any approach, used to determine the type of operations allowed on a view, can be coupled with our extended authorization model.

Although our authorization model allows the specification of negative authorizations for base tables, we do not permit negative authorizations for privileges on views. The reason for this is that supporting negative authorizations on views would make the access control too complex, making it impractical. To see this, suppose that negative authorizations could be specified on views as well. Whenever a user accesses a view V, all the authorizations on all the views directly or indirectly referencing the same tables as V, will have to be checked in order to prevent the user from accessing data denied to him. To illustrate this point further, suppose that views V_1 and V_2 are defined on top of table T. Suppose user A has an authorization for the select privilege on V_1 and, at the same time, a negative authorization for the select privilege on V_2. With respect to the protection requirements, A should not be allowed to select tuples contained in V_2. Since V_1 and V_2 may be not disjoint (i.e., some tuples of T can belong to both of them) the access of A to V_1 must be limited only to those tuples which are not contained in V_2. Therefore, when A accesses V_1, those tuples in V_1 contained also in V_2 should be withheld from A. Enforcing this restriction would entail comparison of the predicates used in defining the views. This approach is obviously not very practical and, therefore, we do not permit negative authorizations to be specified on views.

Negative authorizations are taken into account by allowing the derivation of an authorization for a privilege on a view only if the owner, beside having the authorization for the privilege on all the tables directly referenced by the view, does not have any negative authorization for the privilege on any of the base tables directly or indirectly referenced by the views. Although the user may have multiple authorizations for a privilege on a table, only one authorization for the privilege on the view will be derived. If the user defining the view is authorized for a privilege on all the tables directly referenced by the view with the grant option, the user will also be given the privilege with the grant option on the view. In this case, two authorizations for the privilege on the view are derived, one with the grant option and one without the grant option.[8] The grantor of the authorizations derived on the view is the view owner himself. The timestamp is the time of the view definition.

The relationship between authorizations on views and authorizations on the underlying tables from which the authorizations on the view are derived, is formalized by the following definition.

DEFINITION 14. (Derived Authorization) *Given authorizations* $a_i, a_j \in AS$, a_j is *directly derived* from a_i *(written* $a_i \mapsto a_j$*) iff* $subject(a_i) = subject(a_j) = grantor(a_j)$, $priv(a_i) = priv(a_j)$, $priv\text{-}type(a_i) = priv\text{-}type(a_j) =$ "+", $table(a_i) \rightarrow table(a_j)$, $ts(a_i) < ts(a_j)$, $grant\text{-}opt(a_i) \geq grant\text{-}opt(a_j)$,[9] and $grantor(a_j)$ is the owner of $table(a_j)$.

a_j is *derived from* a_i *(written* $a_i \Mapsto a_j$*) iff* $a_i \mapsto a_j$ *or* $\exists a_1, \ldots a_n \in AS$, $n \geq 1$, *such that* $a_i \mapsto a_1 \mapsto \ldots \mapsto a_n \mapsto a_j$.

If $a_i \Mapsto a_j$, we say that *derivation relationship* exists between a_i and a_j.

Derived authorizations are graphically represented by an arc connecting the node representing the user and a table to the node representing the user and the view. The timestamp on the arc is the time of the view definition. As an example, consider the authorization state of Fig. 7 and suppose that, user B defines view V_1 at time 40, view V_2 at time 50, view V_3 at time 60, and view V_4 at time 70, as illustrated in Fig. 6. Moreover, suppose that, at time 80, B

7. Although we have assumed that authorizations are specified on whole tables, note that in general the only exception to this rule is the update privilege which can specify single columns. A user can be authorized for the update privilege on a column of a view if he has the update privilege on the corresponding column in the table on which the view is defined.

8. The reason for adding two authorizations is to simplify the definition of the revoke function. Suppose we add only one authorization with the grant option, and suppose that the user's grant option is taken away from one of the underlying tables. In this case, we would need to change his authorization on the view by taking away the grant option. By contrast, if we add two authorizations, we can express the revoke operations as previously defined as deletion of all authorizations for which there are no authorization chains.

9. We consider "yes" > "no".

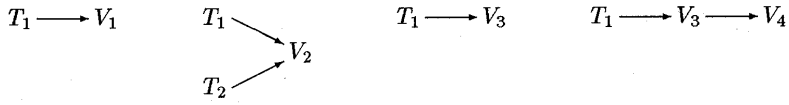


Fig. 6. Derivation graphs of views V_1, V_2, V_3, and V_4.

grants the select privilege on V_4 to user C, and at time 90, B grants the select privilege on V_2 to user D. The authorization state resulting after the execution of all these operations is illustrated in Fig. 8. The direct derivation relationship between the authorizations are as follows:

$a_3 \mapsto \bar{a}_4$, $a_3 \mapsto a_4$, $a_3 \mapsto \bar{a}_5$, $a_3 \mapsto a_5$, $a_3 \mapsto \bar{a}_6$, $a_3 \mapsto a_6$, $a_2 \mapsto \bar{a}_5$, $a_3 \mapsto a_5$, $\bar{a}_6 \mapsto \bar{a}_7$, $a_6 \mapsto \bar{a}_7$, $a_6 \mapsto a_7$.

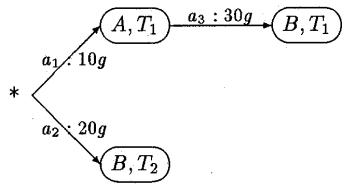


Fig. 7. An authorization state.

We restate the definition of authorization chain by including the derivation relationship as follows.

DEFINITION 15. (Authorization Chain) *An authorization chain is a sequence* $\langle a_1, a_2, \ldots, a_n \rangle$ $(n \geq 1)$ *of authorizations such that* $a_1, a_2, \ldots, a_n \in AS$, a_1 *rel* a_2 *rel* $\ldots$ *rel* a_n, *where rel* $\in \{\rightarrow, \mapsto\}$ and $g(a_1) =$ "*".

Authorizations in an authorization chain can be defined on different tables. Given an authorization chain $ch \in C(AS)$, the *coverage* of *ch*, denoted with *cov*(*ch*), is a sequence of tables defined as follows: $cov(ch) = \langle t_1, \ldots, t_n \rangle$ such that $t_i \rightarrow t_{i+1}$ and $\forall a_i \in AS : a_i \in ch \Rightarrow table(a_i) \in cov(ch)$. For instance, with reference to Fig. 8, $cov(\langle a_1, a_3, a_6, a_7, a_8 \rangle) = \langle T_1, V_3, V_4 \rangle$.

Given these definitions, we represent the constraint that the owner of a view can own an authorization for a privilege on the view only if he owns the authorization for the privilege on all the tables directly referenced by the view by extending the Connectivity Property as follows.

PROPERTY 3. (Connectivity) An authorization state *AS* satisfies the Connectivity Property iff for every authorization $a \in AS$, and every derivation path of *table*(*a*) there exists an authorization chain covering the path. Formally:

$\forall a \in AS, dp \in DP(table(a)) : \exists \langle a_1, \ldots, a_n \rangle \in C(AS), n \geq 1,$ such that $a = a_n$ and $cov(\langle a_1, \ldots, a_n \rangle) = dp$.

Note that if all authorizations refer to base tables only this property reduces to Property 1, by observing that for any base table *t*, $DP(t) = \{\langle t \rangle\}$.

If a user receives the authorization for a privilege on a view with the grant option, he can grant other users the privilege, possibly with the grant option, on the view. In order for these users to exercise their privileges on the view, it is not necessary that they be authorized for the privilege on the tables directly referenced by the view.

Note that it is possible for a user to have, at the same time, a positive authorization for a privilege on a view and a negative authorization for the same privilege on a base table referenced by the view. In the case of a view owner, this can occur if the negative authorization was granted after the owner defined the view. For other users, this can occur if they independently receive the authorization on the view and the negative authorization on some base table. In our model, negative authorizations always take precedence over positive authorizations. Therefore, if a user has a negative authorization for a privilege on a base table, he will not be able to exercise the privilege on any view referencing the table. Note that this approach is consistent with the use of views as a mechanism for enforcing content-dependent authorizations. To formalize this, we extend the definition of blocked authorization as follows.

DEFINITION 16. (Blocked Authorization) *An authorization* $a_i \in AS$ with $grantor(a_i) \neq$ "*", and with *priv-type*$(a_i) =$ "+", is said to be *blocked* for user $u = subject(a_i)$ iff there exists another authorization $a_j \in AS$ such that $subject(a_i) = subject(a_j)$, $priv(a_i) = priv(a_j)$, *priv-type*$(a_j) =$ "–", $(table(a_i) = table(a_j)$ *or* $table(a_i) \twoheadrightarrow table(a_j))$.

Authorization a_j *is called a blocking authorization for* a_i *for u, written* $a_j \dashv_u a_i$.

Given this definition, we represent the requirement that no authorization for a privilege can be derived for the owner of a view if he owns a negative authorization for the privilege on any table referenced by the view by extending the Blocking Conformity Property as follows.

PROPERTY 4. (Blocking Conformity) *An* authorization state *AS* satisfies the Blocking Conformity Property iff

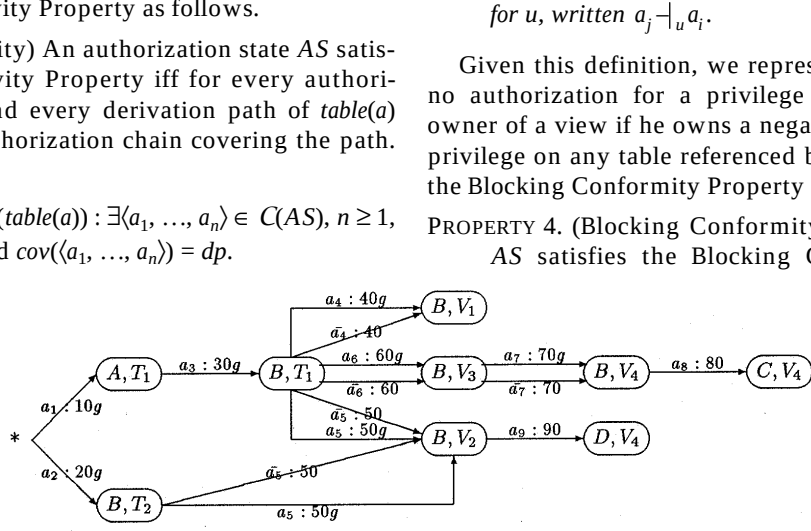


Fig. 8. An authorization state consisting of base tables and views.

$\forall a_i, a_j \in AS, a_i \rightarrow a_j$ *or* $a_i \mapsto a_j : \exists a_z, a_z \dashv_u a_i$, u = *subject* $\Rightarrow bt(a_i, u) > ts(a_j)$.

If, after having defined a view, the view owner receives additional privileges on the underlying tables, the owner will not receive authorizations for them on the view. By contrast, if after having defined a view, a privilege of the view owner on any of the underlying tables is revoked, that privilege may be revoked on the view also (since according to the Connectivity Property the authorizations on the view no longer hold). Therefore, the authorizations of the view owner on the view can only decrease due to revocation of the privileges on the underlying tables, and never increase.

6.2 Extension of the Revoke Operations on Views

6.2.1 Cascading Revocation and Views

According to the semantics of cascading revocation, in addition to the authorizations the revokee may have granted, the authorizations he may have acquired on views referencing the table may need to be revoked. The revoke function with cascade extended to the consideration of views is formalized as follows.

DEFINITION 17. (Revoke Function with Cascade) *Given authorization state AS and a request for revocation* $\langle y, p, t, x\rangle$, $y \in S, p \in P, t \in T, x \in U$, *cascade-rvk*$(AS, \langle y, p, t, x\rangle)$ *generates a new authorization state* $\overline{AS}$ *defined as follows.* Let *REV and CREV be the sets:*

$REV = \{ a \in AS \mid subject(a) = y, priv(a) = p, priv\text{-}type(a) = \text{“+”}, table(a) = t, grantor(a) = x\}$

$CREV = \{ a \in AS \mid \exists dp \in DP(table(a)), \langle a_1, \ldots, a_n, a\rangle \in C(AS), cov(\langle a_1, \ldots, a_n, a\rangle) = dp, \exists a_i \in REV, 1 \le i \le n\}$.

Then, $\overline{AS} = AS - REV - CREV$.

In the above definition, *REV* denotes the set of authorizations being revoked, i.e., the authorizations for the privilege on the table granted by the revoker to the revokee. *CREV* denotes the set of all authorizations *a* for which, after deletion of the authorizations being revoked, there would exist a derivation path of *table*(*a*) which is not covered by any authorization chain. *CREV* is the set of authorizations which would not have existed if the authorizations being revoked (i.e., *REV*) had never been granted.

EXAMPLE 6. Consider the authorization state of Fig. 8 and suppose that user *A* revokes the select privilege on table T_1 from user *B*. The authorization to be revoked is a_3, i.e., REV= $\{a_3\}$. Thus, *CREV* = $\{a_4, \bar{a}_4, a_5, \bar{a}_5, a_6, \bar{a}_6, a_7, \bar{a}_7, a_8\}$. The resulting authorization state is as illustrated in Fig. 9.

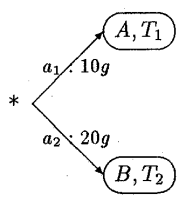


Fig. 9. Authorization state after a_3 is revoked with cascade from Fig. 8.

6.2.2 Noncascading Revocation and Views

According to the semantics of the noncascading revocation, if user *x* revokes a privilege on a table from user *y*, the authorizations for the privilege on the table granted by *y* using the authorizations being revoked are respecified with *x* as the grantor. If *y* granted the privilege with conditions, i.e., by defining a view, *x* should become the grantor of the authorizations for the privilege on the view granted by *y*. With respect to the Connectivity Property, in order for *x* to appear as grantor of any authorization for the privilege on a view, *x* must own the authorization for the privilege on the view with the grant option. Therefore, if *x* needs to become the grantor of any authorization on the view, the authorization for the privilege on the view with the grant option must be given to *x*.[10] The timestamp of the authorization is the definition time of the view, the grantor is *x* himself. To avoid the authorization derived for *x* on the view to be deleted if *y* drops the view, *x* is also considered as an owner of the view. Therefore, a view can have more than one owner, and any of them can drop the view. The view will be actually dropped when the last one of its owners asks for its deletion.

However, the user requiring revocation of a privilege on a table may not always be able to receive the authorization on the views defined on the table. According to the Connectivity Property, if the view is defined on multiple tables, *x* receives the privilege on the view with the grant option, only if he has the authorization for the privilege, with the grant option and with timestamp smaller than the view definition time, on all the tables directly referenced by the view. Moreover, according to the Blocking Conformity Property, *x* must not have any negative authorization for the privilege on any of the base tables referenced by the view with timestamp smaller than the view definition time.

Therefore, absence of positive authorizations, or the presence of negative authorizations on a table referenced by a view, affects the possibility of the user requiring revocation to appear as grantor of the authorizations on the view.

Let $\langle y, p, t, x\rangle$ be a request of user *x* to revoke privilege *p* on table *t* from *y*. The revoke function is based on the definition of the following sets:

- *REV*, the set of authorizations being revoked, i.e., the authorization for *p* on *t* granted by *x* to *y*.
- *SUP*, the set of authorizations for *p* on *t* to be specified with *x* as grantor.
- *POT-DER*, all possible authorizations which could need to be derived for *x* on views defined by *y*. They are obtained from authorizations derived for *y* from some authorization being revoked.
- *DER*, the subset of authorizations in *POT-DER* which satisfies the Connectivity and the Blocking Conformity Properties, i.e., which can be derived from *x*'s authorizations (either already in *AS* or being derived), and which, according to the authorizations in *AS*, were not blocked at the time of the definition of the considered view.

10. Since *x* is given the authorization on the view only to allow him to appear as grantor of the authorizations on the view, the only authorization for the privilege being revoked with the grant option is given to him.

- *SDER*, the set of authorizations on views defined on t to be specified with x as grantor.

Given an authorization state *AS* and a request for revocation of privilege p on table t from user y by user x, the revoke function determines which authorizations on the table must be revoked (*REV*) and which must be specified with the revoker as grantor (*SUP*). Then, the views referencing the table are considered to determine the authorization which can be derived for the revoker (*POT-DER*). The authorizations which can be actually derived for the revoker (*DER*) and which have to be specified with the revoker as grantor (*SDER*) are determined. The authorizations to be specified with the revoker as grantor (*SUP* and *SDER*) and the authorizations to be derived (*DER*) which either supports some authorizations in *SDER* or are needed to derive some authorizations which supports authorization in *SDER* are added to the authorization state. The cascade revoke function is then applied to the authorization state so obtained. The revoke function without cascade is formally defined as follows.

DEFINITION 18. (Revoke Function) *Given authorization state AS and a request for revocation* $\langle y, p, t, x\rangle$, *with* $y \in S$, $p \in P$, $t \in T$, $x \in U$, $rvk(AS, \langle y, p, t, x\rangle)$ *generates a new authorization state* $\overline{AS}$ *defined as follows. Let REV, SUP, POT-DER, DER, and SDER be the following sets of authorizations:*

$$REV = \{ a \in AS \mid subject(a) = y, priv(a) = p, priv\text{-}type(a) = \text{“+”}, table(a) = t, grantor(a) = x\}$$

$$SUP = \{ \langle subject(a_i), priv(a_i), priv\text{-}type(a_i), table(a_i), ts(a_i), x, grant\text{-}opt(a_i)\rangle \mid \exists a_i \in AS, \exists a_j \in REV : subject(a_i) \neq x, subject(a_i) \neq y, a_j \rightarrow a_i\}.$$

$$POT\text{-}DER = \{ \langle x, priv(a_i), priv\text{-}type(a_i), table(a_i), ts(a_i), x, grant\text{-}opt(a_i)\rangle \mid \exists a_i \in AS, \exists a_j \in REV: grant\text{-}opt(a_i) = \text{“yes”}, a_j \mapsto\!\!\!\!\rightarrow a_i\}$$

$$DER = \{ a_i \in POT\text{-}DER \mid \forall dp \in DP(table(a_i)), \exists ch \in C(AS \cup SUP \cup POT\text{-}DER), \nexists a_b \in AS : ch = \langle a_1, \ldots, a_k\rangle, a_1, \ldots, a_k \notin REV, a_k \mapsto a_i, cov(\langle ch, a_i\rangle) = dp, a_b \dashv_x a_k, bt(a_k, x) < ts(a_i)\}$$

$$SDER = \{ \langle subject(a_i), priv(a_i), priv\text{-}type(a_i), table(a_i), ts(a_i), x, grant\text{-}opt(a_i)\rangle \mid \exists a_i, a'_j \in AS, a_j \in DER, \nexists a_b \in AS : subject(a_j) = x, subject(a'_j) = grantor(a'_j) = y, priv(a'_j) = priv(a_j), table(a'_j) = table(a_j), ts(a'_j) = ts(a_j), subject(a_i) \neq x, subject(a_i) \neq y, a'_j \rightarrow a_i, a_b \dashv_x a_j, bt(a_j, x) < ts(a_i)\}$$

$$\overline{AS} = AS \cup SUP \cup SDER \cup \{a_j \in DER \mid \exists a_i \in SDER, a_k \in DER : a_j \mapsto\!\!\!\!\rightarrow a_k \rightarrow a_i \vee a_j \rightarrow a_i\}$$

Then $\overline{AS} = cascade\text{–}rvk(\overline{AS}, \langle y, p, t, x\rangle)$.

EXAMPLE 7. Consider the authorization state illustrated in Fig. 8 and suppose that user A revokes the select privilege on table T_1 from user B. As a result B loses the authorizations derived for him on views V_1, V_2, V_3, V_4. The authorization granted by B to C on V_4 is respecified with A as grantor and, therefore, A becomes owner of V_3 and V_4 and is given the authorization for the select privilege with the grant option on these views. No authorization is derived for A on V_1 since there is no authorization granted on V_1. Similarly, no authorization is derived for A on V_2 since A does not have the necessary authorization on T_2. The revocation process works as follows. The authorization to be revoked is authorization a_3, i.e., $REV = \{a_3\}$. No authorization is supported by the authorization being revoked, $SUP = \emptyset$. The set of potential authorizations to be derived for the revoker is $POT\text{-}DER = \{a'_4, a'_5, a'_6, a'_7\}$, where a'_4, a'_5, and a'_6 are equal to a_4, a_5, a_6, and a_7, respectively (i.e., all authorizations with the grant option derived for B) but with A as subject and grantor. The set of authorizations in *POT-DER* which can actually be derived is then determined as $DER = \{a'_4, a'_6, a'_7\}$. Authorization a'_5, belonging to *POT-DER*, does not belong to *DER*, since it cannot be derived given that A does not own any authorization for the select privilege on T_2. The set of authorizations on views to be restated with A as grantor is then determined as $SDER = \{a'_8\}$, where a'_8 is equal to a_8 but with A as grantor. Then, the authorizations to be specified with the revoker as grantor (*SUP* and *SDER*) and the authorizations in *DER* needed to support authorization in *SDER* or to derive authorizations supporting some authorization in *SDER* (a'_6, a'_8) are added to the authorization state. Finally, cascade revocation is applied. The application of function *cascade-rvk* determines $REV = \{a_3\}$, and $CREV = \{a_4, \overline{a}_4, a_5, \overline{a}_5, a_6, \overline{a}_6, a_7, \overline{a}_7, a_8\}$. The resulting authorization state is as illustrated in Fig. 10.

7 GROUPS

In the extended model [25], groups can appear as subjects of authorizations. Members of groups can be users as well as other groups. A user can exercise any authorization that has been given to the groups to which he belongs. When a user grants an authorization by using the grant option of some group to which he belongs, an additional authorization (called *connecting* authorization), with the group as grantor and the user as subject, may be added to the authorization state. Revocation of authorizations from groups and from users requires special handling and may require the insertion of new connecting authorizations.

Unfortunately, the approach proposed by Wilms and Lindsay is inadequate for the noncascading revocation and the negative authorizations [5]. Instead of adding connecting authorizations, we represent the relationship between authorizations of a group and authorizations that users of the group may have granted or acquired (on views) from them by extending the definitions of supporting and derived authorizations.

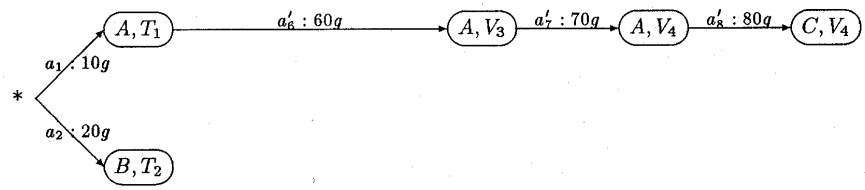


Fig. 10. Authorization state after a_3 is revoked without cascade from Fig. 8.

7.1 Group Membership

As in [25], we also assume that members of groups can be both users as well as other groups and groups need not be disjoint. A timestamp is associated with the membership of a subject (user or group) to a group stating the time the subject became a member of the group. In the following, *tm*(*m*, *G*) indicates the time associated with the membership of *m* to group *G*, where *m* is either a user or a group. Membership of a user *u* to a group *G* is *direct*, if the user is defined as a member of *G*. It is *indirect*, if the user is a member of a group G_i, which is member, either direct or indirect, of *G*. Therefore, a user can belong to a group in multiple ways, either as a direct or an indirect member. For example, suppose *C* belongs to groups G_1 and G_2 which are both members of a same group G_3. Then, *C* indirectly belongs to G_3 both as a member of G_1 and as a member of G_2.

The membership relation between users and groups is formalized by the following definition.

DEFINITION 19. (Membership) *Given a user u and a group G, u belongs directly to group G, written* $u \in G$, *if u is a member of G. User u belongs to G, written* $u \in\in G$, *iff there exists a sequence of subjects* $\langle s_1, \ldots, s_n \rangle$, $n > 1$, *such that* $s_1 = u$, $s_n = G$, *and* $s_i \in s_{i+1}$, $1 \leq i \leq n-1$. *Sequence* $\langle s_1, \ldots, s_n \rangle$ *is called a membership path of u to G, written mp*(*u*, *G*).

In the following, *MP*(*u*, *G*) indicates all the membership paths between user *u* and group *G*.

The configuration of groups can be graphically illustrated with a graph whose nodes represent users and groups. An arc labeled *t* directed from node s_i to node s_j indicates that s_i became a direct member of s_j at time *t*, i.e., $tm(s_i, s_j) = t$. A membership path between a user and a group is, therefore, a path connecting the user and the group in the group configuration graph.

EXAMPLE 8. Consider users *A*, *B*, *C*, *D*, and *E*. At time 25, group G_1 is defined and its members are users *A* and *B*. At time 40, group G_2 is defined and its members are users *C* and *D*. At time 60, group G_3 is defined and its members are groups G_1 and G_2. At time 80, user *C* is added to group G_1. Finally, at time 100, user *E* is added to group G_2. The group configuration graph is illustrated in Fig. 11.

Since a user can belong to a group indirectly, the membership time of a user to a group depends on the time on all the arcs in the membership paths from the user to the group. The following definition characterizes the membership time through a path.

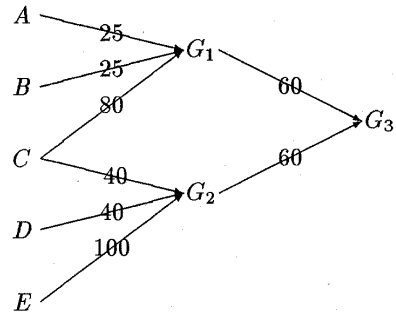


Fig. 11. A group's configuration graph.

DEFINITION 20. (Time of a Membership Path) *Given a membership path* $mp(u, G) = \langle s_1, \ldots, s_n \rangle$, *the time of the path, written* $ts(mp(u, G))$, *is defined as the maximum time along the path. Formally:*

$$ts(mp(u, G)) = max(\{tm(s_i, s_{i+1}) \mid i = 1, \ldots, n-1\}).$$

For instance, with reference to the group's configuration graph illustrated in Fig. 11, $ts(\langle C, G_1, G_3 \rangle) = 80$, and $ts(\langle C, G_2, G_3 \rangle) = 60$.

The membership time of a user to a group can then be defined as follows.

DEFINITION 21. (Membership Time) *Given a user u and a group G such that* $u \in\in G$, *the membership time of u to G, denoted by ts*(*u*, *G*), *is the minimum time among those of all the membership paths from u to G. Formally:*

$$ts(u, G) = min(\{ts(mp) \mid mp \in MP(u,G)\}).$$

For instance, with reference to the group's configuration graph illustrated in Fig. 11, $ts(C, G_3) = \min(\{ts(\langle C, G_1, G_3 \rangle), ts(\langle C, G_2, G_3 \rangle)\}) = 60$.

7.2 Authorizations of Users and Groups

A user is allowed to exercise, beside his own authorizations, all the authorizations of any group to which he belongs, regardless of whether they have been granted before or after the membership time of the user to the group. However, the user cannot use the authorizations of a group before joining the group itself.[11] Therefore, the time at which a user can actually use an authorization of a group is the maximum between the membership time of the user to the group and the time at which the authorization was granted. This is formalized by the following definition.

DEFINITION 22. (Authorization Actual Time) *Given a user u*

11. Although this may seem a trivial observation it is important to keep it in mind when considering revocation, where all the authorizations the revokee may have, either personal or as a member of a group, have to be considered to determine which of the authorizations he granted have to be revoked.

and an authorization $a \in AS$, with $u = subject(a)$ or $u \in\in subject(a)$, the actual time of the authorization for user u, denoted by $at(a,u)$ is defined as follows:

$$at(a, u) = \begin{cases} ts(a) & \text{if } u = subject(a) \\ \max(ts(a), ts(u, subject(a))) & \text{otherwise} \end{cases}$$

For instance, consider the groups configuration graph illustrated in Fig.10 and authorization $a = \langle G_1, select, +, T, 70, E, yes\rangle$. The actual times of the authorization for the users belonging to G_1 are as follows $at(a, A) = at(a, B) = 70$, $at(a, C) = 80$.

If a group is authorized for a privilege on a table with the grant option, any user in the group can grant other subjects the privilege and the grant option on it. The grantor of these authorizations is the user himself, not the group. The reason for this is that it is important to know which of the users of the group granted the authorization so that that user only, and not any other users in the group, can revoke it. Moreover, this allows revocation of the authorization upon removal of the user from the group. In terms of our formalism, the authorization granted by a user can be supported by authorizations of the groups to which the user belongs. To represent this, we extend the definition of supporting relationship as follows.

DEFINITION 23. (Supporting Authorization) *Given two authorizations a_i, $a_j \in AS$, a_i supports authorization a_j (written $a_i \rightarrow a_j$) iff ($grantor(a_j) = grantor(a_i)$ or $grantor(a_j) \in\in grantor(a_i)$), $priv(a_i) = priv(a_j)$, $table(a_i) = table(a_j)$, $grant\text{-}opt(a_i) =$ "yes," $at(a_i, grantor(a_j)) < ts(a_j)$.*

Note that this definition reduces to Definition 3 if users are considered as the only subjects.

To represent this concept, we extend our graphical notation as follows. Every time a user, belonging to a group owning the authorization for a privilege on a table with the grant option, grants an authorization for the privilege on the table, a node $\langle use, table, privilege\rangle$[12] and an arc into this node from node $\langle group, table, privilege\rangle$ are added. The time on the label of the arc is the actual time of the authorization of the group for the user.[13] To distinguish these nodes and arcs from nodes and arcs corresponding to real authorizations, we represent them with thick lines. Note that we use them only for the sake of clarity in the explanation; they do not correspond to any authorization in the authorization state.

EXAMPLE 9. Consider authorization state AS consisting of the following authorizations:

$$a_1 = \langle C, select, +, T, 10, *, yes\rangle$$
$$a_2 = \langle B, select, +, T, 20, C, yes\rangle$$
$$a_3 = \langle G_1, select, +, T, 30, C, yes\rangle$$
$$a_4 = \langle D, select, +, T, 40, B, yes\rangle$$
$$a_5 = \langle E, select, +, T, 50, A, yes\rangle$$

where the groups are as illustrated in Fig. 11. In particular, A and B have been members of group G_1 since time 25. The authorization state is illustrated in Fig. 12.

12. If such a node already exists, only the arc is added.

13. If more than one authorization exists, several arcs are added, one for each authorization.

Note that arcs labeled 30*g* between node $\langle G_1, T\rangle$ and nodes $\langle B, T\rangle$ and $\langle A, T\rangle$ do not correspond to any authorization. The authorization chains are as follows:

$$C(AS) = \{\langle a_1, a_2, a_4\rangle, \langle a_1, a_3, a_4\rangle, \langle a_1, a_3, a_5\rangle\}.$$

Beside the authorizations for privileges, a user also inherits all the negative authorizations of the groups to which he belongs. Thus, possible negative authorizations of a group to which a user belongs may block other authorizations the user may have either personally or as a member of some group.

To represent this, we extend the definition of blocked authorizations as follows.

DEFINITION 24. (Blocked Authorization) *An authorization $a_i \in AS$ with $grantor(a_i) \neq$ "*", with $priv\text{-}type(a_i) =$ "+", is said to be blocked for user u, with $u = subject(a_i)$ or $u \in\in subject(a_i)$,* iff *there exists an authorization $a_j \in AS$ such* that *($u = subject(a_j)$ or $u \in\in subject(a_j)$), $priv(a_i) = priv(a_j)$, $priv\text{-}type(a_j) =$ "–", ($table(a_i) = table(a_j)$ or $table(a_i) \twoheadrightarrow table(a_j)$). Authorization a_j is called a blocking authorization for a_i for user u, written $a_j \dashv_u a_i$.*

Note that an authorization given to a group may be blocked only for some of its members. The time at which an authorization becomes blocked for a user depends on both the actual time of the authorization and the actual time of all the authorizations blocking it. If either the blocked authorization or any of the blocking authorizations have a group as subject, the blocking time depends on the membership time of the user to the group. To represent this, we extend the definition of blocking time as follows.

DEFINITION 25. (Blocking Time) *Given an authorization $a_i \in AS$, blocked for user u, such that $u = s(a_i)$ or $u \in\in s(a_i)$, the blocking time of a_i for u is defined as the maximum between the actual time of the authorization and the minimum among the actual times of all authorizations blocking a_i for user u. Formally:*

$$bt(a_i, u) = \max(at(a_i, u), \min(\{at(a_k, u) \mid a_k \dashv_u a_i\})).$$

7.3 Group's Authorizations and Views

The fact that a user belongs to a group may give him more authorizations (authorizations derived from the authorizations of the group) or fewer authorizations (in case the group has some negative authorizations). To express the fact that the authorizations of the owner of a view on the view can be derived from authorizations of groups to which the user belong, we extend the definition of derived authorization as follows.

DEFINITION 26. (Derived Authorization) *Given two authorizations $a_i, a_j \in AS$, a_j is directly derived from authorization a_i (written $a_i \mapsto a_j$) iff:*

$$subject(a_j) = subject(a_i) \text{ or } subject(a_j) \in\in subject(a_i), subject(a_j) = grantor(a_j), priv(a_i)$$
$$= priv(a_j), table(a_i) \rightarrow table(a_j), grant\text{-}opt(a_i)$$
$$\geq grant\text{-}opt(a_j), at(a_i, subject(a_j)) < ts(a_j)$$

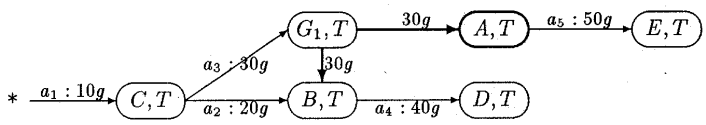


Fig. 12. An authorization state consisting of users and groups (users *A* and *B* have been in group G_1 since time 25).

and grantor(a_j) is the definer of table(a_j); a_j is derived from a_i (written $a_i \mapsto\!\!\!\!\gg a_j$) iff $a_i \mapsto a_j$ or $\exists a_1, \ldots a_n$ such that $a_i \rightarrow a_1 \mapsto \ldots \mapsto a_n \mapsto a_j$.

Definition 26 groups differs from Definition 14 in that condition "*subject*(a_j) $\in\in$ *subject*(a_i)" has been added and the actual time of a_i for *subject*(a_j) has been substituted for the timestamp of a_i.

7.4 Revocation of Authorizations from Groups

In this section, we illustrate the revocation of privileges with consideration of authorizations of groups. Given the extensions to the definitions of supporting and derived authorizations, the revoke functions presented in Sections 4 and 6 do not need to be extended further.

When groups are considered, if a user *x* revokes a privilege on a table from a subject *y*, besides *y*'s authorizations, also the authorizations of the groups to which *y* belongs must be considered in determining the authorizations to be deleted (respecified with *x* as grantor). Indeed, authorizations of groups to which *y* belongs may support authorizations that *y* granted which therefore should not be deleted. Moreover, if *y* is a group, all the authorizations that users belonging to the group may have granted or may have acquired on views may also need to be revoked (respecified with *x* as grantor).

EXAMPLE 10. Consider the authorization state of Fig. 12 and suppose that user *C* revokes the select privilege on table *T* from group G_1.

Consider first the revocation with cascade. The authorization to be revoked is a_3, i.e., $REV = \{a_3\}$. Authorization a_5, which, after deletion of a_3 would not belong anymore to any authorization chain must also be deleted, i.e., $CREV = \{a_5\}$. The resulting authorization state is as illustrated in Fig. 13.

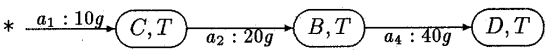


Fig. 13. Authorization state after a_3 is revoked with cascade from Fig. 12.

Consider now the revocation without cascade. The authorization to be revoked is a_3, i.e., $REV = \{a_3\}$. Authorizations a_4 and a_5, granted by *B* and *A*, respectively, after they joined group G_1, must be specified with *C* as the grantor. Thus, $SUP = \{a'_4, a'_5\}$ where a'_4 and a'_5 are equal to a_4 and a_5, respectively, but with *C* as grantor. No authorization was derived from the authorization being revoked, thus $POT\text{-}DER = DER = SDER = \emptyset$. Finally, authorizations in *SUP* are added to the authorization state and recursive revocation is called. Recursive revocation deletes a_3 (*REV*) and a_5 (*CREV*). The resulting authorization state is as illustrated in Fig. 14.

8 Conclusions and Future Work

In this paper we have proposed two extensions to the authorization model proposed by Griffiths and Wade [14], and subsequently extended by Wilms and Lindsay [25]. The effects of these extensions on the execution of operations such as grant and revoke, creation of new tables, and modification of groups configuration have been formally specified.

The proposed authorization model has been implemented. In the implementation, the actual authorization catalogs of a commercial RDBMS have been simulated. To support the advanced features of our authorization model, the structure of these catalogs required only marginal extensions, namely the introduction of two additional column to record the type (positive or negative) of the authorization and whether the authorization is blocked. The goals of the implementation are twofold. On one side, we want to assess the performance of the authorization model and to investigate efficient authorization checking strategies. In the implementation, every time a negative authorization is inserted, all authorization blocked by it are marked. Thus, the control of a request to access a table is performed simply by looking for positive nonblocked authorizations on the table. The configuration of groups is represented by a table maintaining the transitive closure of group membership as proposed by Gagliardi, Lapis, and Linsday [13]. On the other side, we are developing an environment that supports tools for authorization management. We believe that the increased complexity of authorization models, which are required by advanced applications, require in turn such environments. We refer the interested reader to [5] for additional details.

Our model leaves space for further work. In particular, there are more complex authorization management policies that are worth investigating. In our model, negative authorizations always override positive authorizations. The reason is that this is the safest solution whenever there are

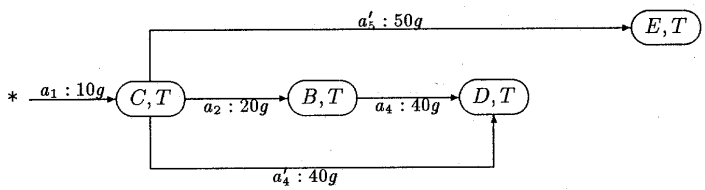


Fig. 14. Authorization state after a_3 is revoked without cascade from Fig. 12.

conflicting authorizations. However, other more sophisticated policies can be considered. We are currently investigating authorization conflict resolution methods based on the users who granted the authorizations, on information explicitly provided by the users, and on the subjects of the authorizations (groups or single users). Another research direction that we are investigating concerns the use of an authorization log mechanism for tracking the history of all authorization commands issued by the users. This log mechanism is particularly useful for noncascading revoke operations since the original grantor of an authorization is replaced by the user performing the noncascading revoke operation and, therefore, the original grantor of the authorization is deleted from the authorization catalog. The log mechanism would be a way to preserve this information. The information in the log can also be used in the framework of temporal authorization mechanisms, which we are currently investigating [4].

ACKNOWLEDGMENTS

The work of Sushil Jajodia was partially supported by the National Science Foundation under Grant No. IRI-9303416 and by the National Security Agency under Contract No. MDA904-94-C-6118. A preliminary version of this paper appeared under the title "Authorizations in Relational Database Management Systems" in the *Proceedings of the First ACM Conference on Computer and Communications Security*, pp. 130–139, November 1993.

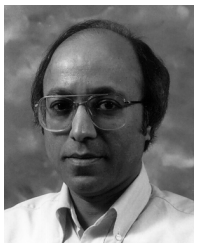
REFERENCES

[1] M.M. Astrahan et al., "System R: A Relational Approach to Database Management," *ACM TODS*, vol. 1, no. 2, pp. 97–137, June 1976.
[2] F. Bancilhon and N. Spyratos, "Update Semantics of Relational Views," *ACM TODS*, vol. 6, no. 4, pp. 557–575, Dec. 1981.
[3] R.W. Baldwin, "Naming and Grouping Privileges to Simplify Security Management in Large Databases," *Proc. IEEE Symp. Security and Privacy*, Oakland, Calif., pp. 116–132, Apr. 1990.
[4] E. Bertino, C. Bettini, and P. Samarati, "A Time Based Authorization Model," *Proc. ACM Int'l Conf. Computer and Comm. Security*, Fairfax, Va., pp. 126–135, Nov. 1994.
[5] E. Bertino, P. Samarati, and S. Jajodia, "An Extended Authorization Model for Relational Databases," internal report, extended version, pp. 1-60, 1994.
[6] E. Bertino and L.M. Haas, "Views and Security in Distributed Database Management Systems," *Proc. First Int'l Conf. Extending Database Technology (EDBT)*, Venice, Italy, *Lecture Notes in Computer Science*, vol. 303, pp. 155–169, Springer-Verlag, 1988.
[7] V. Brosda and G. Vossen, "Update and Retrieval in a Relational Database through a Universal Schema Interface," *ACM TODS*, vol. 13, no. 4, pp. 449–485, Dec. 1988.
[8] D.D. Chamberlin et al., "A History and Evaluation of System R," *Comm. ACM*, vol. 24, no. 10, pp. 632–646, 1981.
[9] S. Cosmadakis and C.H. Papadimitriou, "Updates of Relational Views," *J. ACM*, vol. 31, no. 4, pp. 742–760, Oct. 1984.
[10] Department of Defense, *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, Dec. 1985.
[11] N. Gal-Oz, E. Gudes, and E.B. Fernandez, "A Model of Methods Access Authorization in Object-Oriented Databases," *Proc. Int'l Conf. Very Large Data Bases*, Dublin, Ireland, pp. 52–61, Aug. 1993
[12] R. Fagin, "On an Authorization Mechanism," *ACM TODS*, vol. 3, no. 3, pp. 310–319, Sept. 1978.
[13] R. Gagliardi, G. Lapis, and B.G. Lindsay, "A Flexible and Efficient Database Authorization Facility," IBM Research Report RJ6826, 1989.
[14] P.G. Griffiths and B. Wade, "An Authorization Mechanism for a Relational Database System," *ACM TODS*, vol. 1, no. 3, pp. 242–255, Sept. 1976.
[15] *Informix-OnLine/Secure Security Features User's Guide*. Informix Software Inc., Menlo Park, Calif., Apr. 1993.
[16] R. Langerak, "View Updates in Relational Databases with an Independent Scheme," *ACM TODS*, vol. 15, no. 1, pp. 40–66, Dec. 1990.
[17] *Oracle7 Server Administrator's Guide*. Oracle Corp., Redwood City, Calif., Dec. 1992.
[18] "Database Language SQL2," (ISO/ANSI working draft), J. Melton, ed., ANSI X3H2-90-309, Aug. 1990.
[19] T.F. Lunt, "Access Control Policies for Database Systems," *Database Security, II: Status and Prospects*, C.E. Landwehr, ed., North-Holland: Elsevier Science Publisher B.V., pp. 41–52, 1989.
[20] T.F. Lunt et al., "Secure Distributed Data Views, vol. 1–4," SRI International, Palo Alto, Calif, 1989.
[21] F. Rabitti, E. Bertino, W. Kim, and D. Woelk, "A Model of Authorization for Next-Generation Database Systems," *ACM TODS*, vol. 16, no. 1, pp. 88–131, Mar. 1991.
[22] R. Sandhu and P. Samarati, "Access Control Principles and Practice," *IEEE Comm.*, pp. 40–48, Sept. 1994.
[23] M. Satyanarayanan, "Integrating Security in a Large Distributed System," *ACM-TOCS*, vol. 7, no. 3, pp. 247-280, Aug. 1989.
[24] P.G. Selinger, "Authorizations and Views," *Distributed Data Bases*. I.W. Draffan and F. Poole, eds., Cambridge, Mass.: Cambridge Univ. Press, pp. 233–246, 1980.
[25] P.F. Wilms and B.G. Lindsay, "A Database Authorization Mechanism Supporting Individual and Group Authorization," *Distributed Data Sharing Systems*. R.P. van de Riet and W. Litwin, eds., North-Holland: Elsevier Science, pp. 273–292, 1982.

Elisa Bertino (M'88) is professor of computer science in the Department of Computer Science of the University of Milan, where she heads the Database Systems Group. She has also been on the faculty in the Department of Computer and Information Science of the University of Genova, Italy. Until 1990, she was a researcher for the Italian National Research Council in Pisa, Italy, where she headed the Object-Oriented Systems Group. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, California, at the Microelectronics and Computer Technology Corporation in Austin, Texas, and at George Mason University.

Her main research interests include object-oriented databases, distributed databases, deductive databases, multimedia databases, interoperability of heterogeneous systems, integration of artificial intelligence and database techniques, and database security. In those areas, Prof. Bertino has published several papers in refereed journals, such as *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, *Acta Informatica*, *Information Systems*, and in proceedings of international conferences and symposia. Prof. Bertino is co-author of the book *Object-Oriented Database Systems—Concepts and Architectures*, Addison-Wesley International, 1993, and co-author of the forthcoming book *Intelligent Database Systems*, Addison-Wesley International. She has participated in several research projects sponsored by the Italian National Research Council and the European Economic Communities. She is on the editorial board of *IEEE Transactions on Knowledge and Data Engineering*, and *the International Journal of Theory and Practice of Object Systems*. She is serving as program co-chair for the 1998 IEEE International Conference on Data Engineering. She is a member of the IEEE.

Pierangela Samarati received the doctoral degree in computer science from the University of Milan, Italy, in 1988. She is now an assistant professor of computer science at the University of Milan. Her main research interests are information systems security, database security, authorization models, and databases. On these topics, she has published several papers. She has been a visiting researcher at Stanford University, California, and at George Mason University, Virginia. She is currently serving as the the Italian representative in the IFIP TC-11 (Technical Committee 11 on Security and Protection in Information Processing Systems). She is co-author of the book *Database Security*, Addison-Wesley, 1995.



Sushil Jajodia (M'88–SM'88) received his PhD degree from the University of Oregon, Eugene. He is professor of information and software systems engineering and director of the Center for Secure Information Systems at the George Mason University, Fairfax, Virginia. He joined GMU after serving as the director of the Database and Expert Systems Program within the Division of Information, Robotics, and Intelligent Systems at the National Science Foundation. Before that, he was head of the Database and Distributed Systems Section in the Computer Science and Systems Branch at the Naval Research Laboratory, Washington and associate professor of computer science and director of graduate studies at the University of Missouri, Columbia.

Dr. Jajodia's research interests include information security, temporal databases, and replicated databases. He has published more than 150 technical papers in the refereed journals and conference proceedings and has edited or co-edited 10 books, including *Advanced Transaction Models and Architectures*, Kluwer (1997), *Multimedia Database Systems*: *Issues and Research Directions*, Springer-Verlag Artificial Intelligence Series (1996), *Information Security*: *An Integrated Collection of Essays*, IEEE Computer Society Press (1995), and *Temporal Databases Theory*, *Design, and Implementation*, Benjamin/Cummings (1993). He received the 1996 Kristian Beckman award fro IFIP TC 11 for his contributions to the discipline of Information Security.

Dr. Jajodia has served in different capacities for various journals and conferences. He is the founding co-editor-in-chief of the *Journal of Computer Security.* He is on the editorial boards of *IEEE Concurrency and International Journal of Cooperative Information Systems* and a contributing editor of the *Computer & Communication Security Reviews.* He serves on numerous conference program committees including the 1997 IEEE Symposium on Security and Privacy, the 1997 Computer Security Foundations Workshop, and the 1997 Very Large Data Base Conference. He has been named a Golden Core member for his service to the IEEE Computer Society. He is past-chair of the IEEE Computer Society Technical Committee on Data Engineering and the Magazine Advisory Committee. He is a senior member of the IEEE and a member of the IEEE Computer Society and Association for Computing Machinery. The URL for his web page is http://www.isse.gmu.edu/~csis/faculty/jajodia.html.