Authorizations in Relational Database Management Systems

Elisa Bertino* Pierangela Samarati* Sushil Jajodia†

Abstract

This paper proposes two major extensions to the authorization model for System R relational database management system. The first extension concerns the revoke operation. The revised model provides for a new type of revoke operation, called *noncascading* revoke, in addition to the System R *cascading* revoke operation. Unlike cascading revoke, noncascading revoke operation does not recursively remove privileges from users. The second extension concerns negative authorization. The details related to its application are specified in the paper.

1 Introduction

In relational database management systems, access control is usually based on the identity of the users and the rules that specify for each user as well as table in the system the types of accesses (e.g., read, write, or execute) the user is allowed on the table. Whenever a user requests access to a table, the request is checked against the specified authorizations; if a *positive* authorization exists stating that the user can access the table in the specific mode, the access is granted; otherwise it is denied.

Authorizations for privileges on tables are usually administered by the owners of the tables. The user who creates a table becomes the owner of the table. The owner is entitled to execute any privilege on the table and, moreover, can grant (or revoke) other users' authorizations for any privilege on the table. Authorizations can be granted with the grant option. If a user owns an authorization for a privilege on a table with the grant option, he can grant the privilege, and the grant option, to other users.

In this paper, we consider the System R authorization model [5, 6] that enforces above policies, and propose two major extensions to this model. The first extension concerns the revoke operation. Whenever a user revokes a privilege on a table from another user, the revoke operation is applied recursively to existing authorizations: the privilege is taken away not only from the revokee but from all those users who received the privilege through the revokee.

The recursive revocation, though useful in some cases, is not always desirable. For instance, suppose a user changes his job due to a promotion. This change may imply a change in the responsibilities of the user and, therefore, in his privileges. The user may be granted new authorizations, while some of his previous authorizations may have to be revoked. Applying a recursive revocation will result in the undesirable effect of deleting the authorizations the user granted and, recursively, the authorizations granted through them, which will then need to be reissued. All application programs depending on the revoked authorizations will also be invalidated [3]. Many other examples can be found where the effect of recursively deleting the authorizations upon a revoke request is not wanted. Therefore, we propose a new *noncascading* revoke operation, in addition to the usual *cascading* revoke operation. Under noncascading revocation, the privilege is taken away from the revokee, but not from others who received the privilege through the revokee.

The second extension concerns *negative* authorizations. user. If a user has a negative authorization for a privilege on a table, the user can neither exercise nor administer that privilege (i.e., grant or revoke authorizations) on the table. Negation is stronger than absence of authorization. If a user is granted a negative authorization for a privilege on a table, then the user will not be able to exercise a privilege on the table even though he may have received or will receive in the future authorizations giving the privilege. Thus, negative authorizations provide a mechanism to prevent a given user from being able to exercise a privilege on a table. This is particularly important in environments where authorization administration is decentralized and other users, beside the owner of a table, can grant authoriza-

*Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy.

†Center for Secure Information Systems and Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA 22030-4444, U.S.A. The work of S. Jajodia was partially supported by a grant from the National Science Foundation under IRI-9303416.

1st Conf.- Computer & Comm. Security '93-11/93 -VA,USA
© 1993 ACM 0-89791-629-8/93/0011...$1.50

tions on the table and, therefore, a user may obtain the authorization for a privilege on a table against the desire of the table's owner.

The rest of this paper is organized as follows. We begin in the next section with some basic definitions. In the three sections that follow, we describe the proposed extensions and provide the correctness proofs of our algorithms. We conclude the paper with a summary and suggestions for further work.

2 Basic authorization model

An authorization in our model can be expressed as a tuple $\langle s, p, t, ts, g, go \rangle$ where

s is the user to whom the authorization is granted (i.e. the *grantee*);

p is the access mode (i.e., select, insert, delete, or update);

t is the table[1] to which the authorization is referred;

ts is the time[2] at which the authorization was granted;

g is the user who granted the authorization (i.e. the *grantor*);

$go \in$ {yes, no} indicates whether s has the grant option for p on t.

Tuple $\langle s, p, t, ts, g, go \rangle$ states that user s has been granted privilege p on table t by user g at time ts. s is authorized to grant other users privilege p on table t as well as the grant option on it iff $go =$ "yes." 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. Tuple $\langle C, select, T, 20, B, no \rangle$ indicates that user C can select tuples from table T and that this privilege was granted to C by user B at time 20. The authorization does not entitle user C to grant other users the select privilege on T since C does not have the grant option.

In the following, given an authorization a, $s(a)$, $p(a)$, $t(a)$, $ts(a)$, $g(a)$, $go(a)$ denote respectively the grantee, the access mode, the table, the time, the grantor, and the grant option in a. For example, $g(a)$ denotes the grantor of authorization a.

The sequence of grant operations of a privilege on a table can be represented by a labeled graph where each node represents a user. A labeled arc between node u_1 and node u_2 indicates that u_1 granted the privilege on the table to u_2. Every arc is labeled with the *time* at which the privilege was granted and also with the symbol "g" if the privilege was granted with the grant option. An example of a sequence of grant operations is illustrated in Figure 1.

[1] A table is either a base or a view relation.

[2] A timestamp can be represented by a system maintained counter.

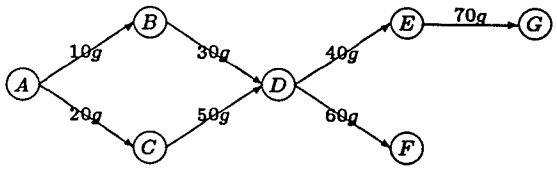

Figure 1: A sequence of grant operations

Every user who has the authorization for a privilege on a table with the grant option, can also revoke the privilege on the table. However, a user can revoke only those authorizations that were granted by him.

The authorizations holding at a given time are the authorizations which have been granted and have not been deleted upon revocation. We refer to the set of authorizations holding at a given time as *authorization state* (AS). Since upon every grant operation a new authorization is inserted, in the following we will refer to grant sequences and authorization states resulting from grant sequences interchangeably.

user a authorizations and which could not not been present are also allowing a privilege to be revoked revocation of the authorizations for the granted.

2.1 Revocation of authorizations

The System R authorization model enforces recursive (or *cascading*) revocation. The semantics of the recursive revocation of privilege p on table t from user y by user x is defined to be as if all the authorizations for p on t granted by x to y had never been granted. Therefore, all the effects brought by the presence of the authorizations being revoked have to be eliminated.

For example, consider the sequence of grant operations for the select privilege on table T shown in Figure 2(a), and suppose that B revokes the privilege on T from C. According to the semantics of recursive revocation, the resulting authorization state has to be as if C had never received the authorization from B. If C had never received the authorization from B, he could not have granted the authorization to D (his request would have been rejected by the system). The sequence of grant requests accepted by the system would have therefore been as in Figure 2(b). Thus, if B revokes the authorization from C, the authorization C granted to D also has to be deleted.

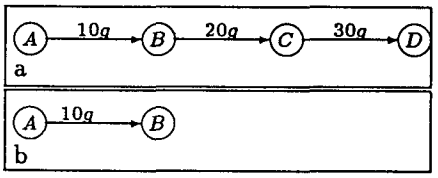

Figure 2: Example of cascading revocation

However, not all the authorizations for the privilege on the table granted by the revokee must be deleted. Indeed, if the revokee received some authorization for the privilege on the table, with the grant option, from some user different from the revoker, some of the authorizations he granted could have been granted anyway. For instance, with reference to the example just mentioned, suppose that C, before granting the authorization to D, received the authorization for the select privilege on T with the grant option from user A as well. In this case even if he had never received the authorization from B, C could have granted the privilege to D. Thus, when B revokes the privilege from C, the authorization granted by C to D does not have to be deleted.

An algorithm implementing the cascading revocation, is shown in Figure 3. The algorithm works as follows. Suppose user x revokes privilege p on table t from user y. All the authorizations for p on t given to y by x are deleted. To determine which authorizations granted by y have to be deleted, all the remaining authorizations of y for privilege p on table t granted with the grant option are considered. Let T_u be the minimum of all timestamps of these authorizations. Then, all authorizations for p on t granted by y to any other user before time T_u are revoked. The process is then repeated for every user whose privilege is revoked.

Algorithm 1
cascading-revoke(revokee, privilege, table, revoker)
/* *revoker* revokes *privilege* on *table* from *revokee* */

begin
  r_tm := current-time;
  casc-revoke(*revokee, privilege, table, revoker, r_tm*)
end

casc-revoke(user, priv, tbl, rev, time)

begin
  /* Delete all the authorizations for *priv* on *tbl* granted */
  /* by *rev* to *user* before *time* */
  delete all authorizations a such that s(a) = *user*,
      p(a) = *priv*,
      t(a) = *tbl*, ts(a) < time, g(a) = *rev from AS*;
  /* Determine T_u the earliest timestamp of the remaining */
  /* authorizations of *user* to grant *priv* on *tbl* */
  T_u := min ({ts(a) | $a \in AS$, s(a) = *user*, p(a) = *priv*,
      t(a) = *tbl*, go(a) = "yes"}, r_tm);
  /* Ask for revocation of the authorizations granted by *user* */
  /* before time T_u */
  for each s_i such that exists a $\in AS$, s(a) = s_i
        p(a) = *priv*, t(a) = *tbl*, g(a) = *user*,
        ts(a) < T_u **do**
    casc-revoke(s_i, *priv, tbl, user,* T_u)
  endfor
end

Figure 3: Cascading revoke algorithm

3 Revocation of authorizations without cascade

In this section, we define the noncascading revoke operation that 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 them, we respecify these authorizations as if they had been granted by the user executing revocation.

The semantics of the revocation without cascade of privilege p on table t from user y by user x has to be as if user x had never granted privilege p on t to user y but, instead, granted all the authorizations[3] for p on t that y granted to other users by using the grant option received from x. Note that this consists in adding the new authorizations with x as grantor to the authorization state, and then applying the recursive revocation.

For instance, consider the grant sequence for the select privilege on table T shown in Figure 4(a)). Suppose now that B revokes the privilege from C. According to the semantics of the revocation, the resulting authorization state has to be as if B had never granted the privilege to C but, instead, granted the authorization given to D by C (grant sequence of Figure 4(b)).

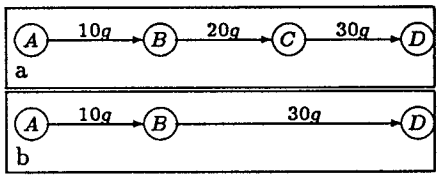

Figure 4: Example of noncascading revocation

Again, as in the case of the recursive revocation, since the revokee (y) may have received the grant option for the privilege on the table from some other users different from the revoker (x), not all the authorizations y granted will be deleted or respecified with x as grantor. In particular, x will be considered as grantor only of the authorizations y granted after receiving the grant option for the privilege on the table from x; y will still be considered as grantor of all the authorizations he granted by using the grant option received from some user different from x.

An algorithm for implementing noncascading revocation is given in Figure 5. The algorithm works as follows. Suppose user x revokes privilege p on table t from user y. The earliest time T_r at which x granted the privilege with the grant option to y is determined. Then, all authorizations for p on t given to y by x are deleted. Let T_u be, as before, the minimum timestamp among those of the remaining authorizations of y for privilege p on table t with the grant option. All the authorizations for

[3] It is important to notice that this applies to all authorizations that y may have granted and that still belong to the authorization state and have not been revoked yet.

p on t granted by y to any other user different from x and y after time T_r are "duplicated" with x as grantor. In this way, x becomes grantor of all authorizations in AS that y granted by using the grant option received from x. Then, all the authorizations for p on t granted by y to any user before time T_u are deleted. The deletion of these authorizations does not imply any further deletion of other authorizations which may have been granted through them. Note that also if the revocation of authorizations is not recursively called to delete all the authorizations which would have not existed if the revokee had never received the authorizations, all invalid authorizations are deleted by the algorithm (see Section 5). Indeed, all authorizations granted by a user who received some authorization from the revokee would have been granted anyway if this user, instead of receiving the privilege with the grant option from the revokee, had received it from revoker, and therefore do not have to be deleted.

Algorithm 2
revoke(revokee, privilege, table, revoker)

```
begin
  r_tm := current-time
  /* Determine T_r earliest time at which revoker granted */
  /* privilege on table with the grant option to revokee */
  T_r := min ({ts(a) | a ∈ AS, s(a) = revokee,
     p(a) = privilege, t(a) = table,
     g(a) = revoker, go(a) = "yes"})
  /* Delete all the authorizations for privilege on table */
  /* granted by revoker to revokee */
  delete all authorizations a such that s(a) = revokee,
        p(a) = privilege, t(a) = table,
        g(a) = revoker from AS
  /* Determine T_u the earliest timestamp of the remaining */
  /* authorizations of revokee for p on t with the grant option */
  T_u := min ({ts(a) | a ∈ AS, s(a) = revokee,
        p(a) = privilege, t(a) = table,
        go(a) = "yes"}, r_tm);
  for each a ∈ AS, p(a) = privilege, t(a) = table,
           g(a) = revoker do
     if ts(a) > T_r, s(a) ≠ revokee, s(a) ≠ revoker
       /* the authorization was granted after receiving */
       /* the authorization from revoker */
       then add ⟨s(a),p(a),t(a),ts(a),x,go(a)⟩ to AS
     endif
     if ts(a) < T_u /* revokee cannot appear as grantor */
                    /* of the authorization any more */
       then delete a from AS
     endif
end
```

Figure 5: Noncascading revoke algorithm

4 Negative authorizations

In our model, negative authorizations can be issued only for access privileges, and not for the administration of the privileges themselves. For example, 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. grant option alone. However, the negative authorization for a privilege on the 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 the select privilege on table T for other users.

Negative authorizations for a privilege on a table can be granted by the owner of the table as well as by any user who owns an authorization for the privilege on the table with the grant option.

To represent negative authorizations, an authorization is now characterized by a tuple $\langle s, p, pt, t, ts, g, go \rangle$ where s, p, t, ts, g, go are defined as before in Section 2 and $pt \in \{+, -\}$, indicating whether the authorization is for a privilege (+) or its negation (−).

Negative authorizations cannot be granted with the grant option; therefore, authorizations with pt = "−" have necessarily go = "no".

To represent grants of negative authorizations, we also extend our graphical notation by adding, to the labels of arcs, the sign (+ or −) of the authorization being granted. Figure 6 illustrates a sequence of grant operations including both positive and negative authorizations.

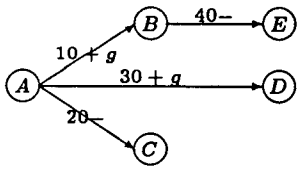

Figure 6: A sequence of grant operations containing positive and negative authorizations

4.1 Authorizations of the user receiving the negative authorization

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

Suppose that a user cannot have at the same time an authorization as well as the negation for a privilege on a table. If a negative authorization for a privilege on a table is granted to a user who has some positive authorizations for the privilege on the table, then either all authorizations for the privilege on the table of the user receiving the negation should be revoked, or the

grant operation for the negative authorization should be refused by the system.

Deleting all positive authorizations for a privilege on a table of a user when the user is granted a negative authorization is not the correct approach. Indeed, user y, receiving a negative authorization for privilege p on table t may have previously received an authorization for p on t from a user different from the user granting the negation. Therefore, deleting all authorizations of y for p on t would imply that x could revoke authorizations not granted by himself. Hence, according to the principle that a user can revoke only the authorizations he granted, we reject this possibility.

On the other hand, the approach of rejecting the insertion of the negative authorization would limit the power of the user authorized to grant negative authorizations.

Therefore, we allow the insertion of a negative authorization without affecting possible positive authorizations the user receiving the negation may have, thus accepting the presence, at the same time, of positive and negative authorizations.

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. Hence, if a user has some positive authorizations for a privilege on a table, these authorizations will not be usable by the user if he has some negative authorization for the privilege on the table. In this case, we say that the positive authorizations are *blocked*. Blocked authorizations are authorizations that cannot be used. Note, however, that blocked authorizations can be revoked.

For example, consider the grant sequence for the select privilege on table T shown in Figure 6. Suppose that at time 50 user B grants a negative authorization for the select privilege on T to user D. Authorization $\langle D, select, -, T, 50, B, no\rangle$ is added. As a result, the authorization granted by A to D becomes blocked.

Though a revocation of a privilege on a table from a user is always desirable before granting the negation for the privilege on the table to the user, the revocation process itself is not automatically executed by the system upon granting the negation. The reason for not automatically enforcing revocation is to leave the user granting the negation the choice of whether requiring for revocation of the privilege with cascade (therefore requiring deletion of all authorizations inserted by using the authorizations being revoked) or without cascade (therefore becoming himself the grantor of all authorizations inserted by using the authorizations being revoked). We discuss this in detail next.

4.2 Authorizations granted by the user receiving the negative authorization

An important issue concerns the authorizations that have been previously granted by a user who receives a negative authorization. If a user receives a negative authorization for a privilege on a table, all his authorizations for the privilege on the table become blocked and the user will not be able to revoke authorizations for the privilege on the table he may have granted. Therefore the problem arises of dealing with the authorizations that the user may have granted before receiving the negation. There are three possible approaches:

1. these authorizations are recursively revoked;
2. negative authorizations are propagated thus blocking these authorizations;
3. these authorizations are neither revoked nor blocked.

Let us examine the various options in more detail. Suppose user x grants a negative authorization for privilege p on table t to user y.

The first solution, i.e., recursively revoking all the authorizations for the privilege on the table granted by the user receiving the negation, is not the correct approach. Again, y may have received some authorization for p on t from users different from x. Therefore, revoking all these authorizations would imply that x can revoke authorizations not granted by him or by using the authorizations he granted to y.

The second approach, i.e., recursively propagating the negation for the privilege on the table also cannot be accepted. The fact that y has to be denied a privilege on a table does not mean that all users who received some authorization for the privilege on the table from y, the users who received some authorization from them, and so on, should also receive a negation for the privilege. Although it may be claimed that if some users received the authorization from y it is not desirable that they continue to use the authorization received, it has to be noticed that they may have received the authorization for p on t from some user different from y and therefore giving them a negative authorization would prevent them from using the authorizations received from the other users. Therefore, we reject this solution.

The last approach, which we adopt, consists in not taking any particular action over the authorizations granted by the user receiving the negation. If the authorizations of y become blocked, it is not so for the authorizations that y granted. If x wishes to block an authorization granted by y to user z, he can always do so by explicitly granting the negation to z.

Therefore, 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 received a negative authorization for p on t, all his authorizations for p on t become blocked. Therefore, the question arises of whether y, after having received the negation for p on t, should be allowed to revoke the authorizations for p on t he granted. The reply is not: even if revoking privileges can only decrease the authorizations in the system, it does not seem appropriate to leave some administrative power on a privilege on a table to a user to whom the privilege on the table is denied.

Note that even if these authorizations cannot be revoked by y, they are not unrevocable. They will either be revoked upon revocation of the blocked

authorizations, or made revocable upon revocation of the blocking authorizations.

For instance, consider the sequence of grant operations illustrated in Figure 7(a) and suppose that, at time 60, user B grants user D a negative authorization for the select privilege on table T. Authorization $\langle D, select, -, T, 60, B, no \rangle$ is added (Figure 7(b)). As a result, the authorization granted by A to D becomes blocked and the authorization granted by D to F is not revocable by user D any more. This situation will be solved upon deletion of either the blocked or the blocking authorization. The first case (deletion of the blocked authorization), will occur if A revokes the select privilege on T from D. The second case (deletion of the blocking authorization) will occur if either B revokes the negation for the select privilege on T from user D or A revokes privilege on T from user B with cascade.

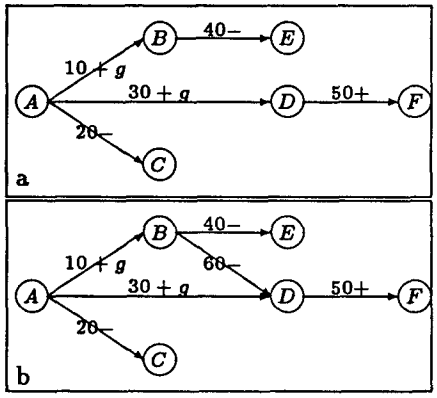

Figure 7: Example of negative authorization

Note that also the user who granted the authorization which became blocked may have received the negative authorization for the privilege on the table considered, and, therefore, a chain of blocked authorizations may exist. However, the fact that the privileges of the owner of a table on the table cannot become blocked, ensures that authorizations are always revocable, even in presence of several blocked authorizations.

4.3 Revocation of authorizations

In this section we discuss the revocation of negative authorizations and extend the algorithms for the revocation of authorizations to the consideration of negative authorizations.

The semantics of the revocation of the negation for a privilege on a table from user y by user x is to eliminate all the negative authorizations for the privilege on the table that x granted to y. Note that the revocation of the negation for a privilege does not bring the authorization state to be as if the authorizations being revoked had never been granted. Indeed, the presence of the negative authorizations being revoked may have had the effect of not allowing grant requests that the revokee may have submitted to the system. Therefore, if negative authorizations being revoked had not been present, some authorizations could have been added to the authorization state and were not.

Upon revocation of the negation for a privilege, producing the authorization state which would have resulted if the negative authorizations being revoked had never been granted, would imply reconsidering all the grant requests rejected by the system submitted after the revokee received the negative authorizations, and possibly inserting the authorizations whose grant would have been accepted if the negative authorizations being revoked had never been granted. For example, consider the grant sequence illustrated in Figure 7(b) and suppose that, at time 70, user D requires to grant user G the select privilege on T. Since D owns a negative authorization for the select privilege on T, his request is rejected by the system, and therefore no authorization is inserted. Suppose now that, at time 80, user B revokes the negation for the select privilege on T from D. Producing the authorization state which would have resulted if B had never granted the negation for the privilege to D would require the insertion of the authorization for G whose grant was required by D at time 70 and was rejected by the system. This approach is obviously not suitable.

The revocation of negative authorizations is very simple. The revocation of the negation for privilege p on table t from user y by user x has the effect of deleting all the negative authorizations for p on t that x granted to y. Since no insertion of further authorizations could have been caused by the presence of the negative authorizations, there is no need to propagate the effect of the revocation.

The consideration of negative authorizations also requires change in the semantics of revocation of privileges. The semantics of the cascading revocation of privilege p on table t from user y by user x was defined to be as if x had never granted p to y. Indeed, if y was authorized for p on t with the grant option, he may have granted other users a negative authorization for p on t. The presence of this negative authorization may have caused the rejection of grant requests submitted by the user who received the negative authorization. Therefore, upon revocation of the privilege from y by x, producing the authorization state that would have resulted if y had never received the privilege from x would require the reconsideration of the grant requests which have been rejected by the system and the possible insertion of the authorizations that would have been accepted. As we have already explained, this solution is obviously not suitable. Therefore, we change the semantics of the revoke operation.

The new definition of semantics of cascading revocation of privilege p on table t from user y by user x is to eliminate from AS all the authorizations (either positive or negative) which could have never existed if x had never granted the privilege to y. An algorithm implementing the revocation of privileges with cascade extended to the consideration of negative authorizations is illustrated in Figure 8.

The semantics of revocation without cascade is

Algorithm 3
cascade-revoke(revokee, privilege, table, revoker)
/* *revoker* revokes *privilege* on *table* from *revokee* */

begin
r_tm := current-time;
casc-revoke(*revokee, privilege, table, revoker, r_tm*)
end

casc-revoke(user, priv, tbl, rev, time)

begin
/* Delete all the authorizations for *priv* on *tbl* granted */
/* by *rev* to *user* before *time* */
delete all authorizations a such that s(a) = *user*,
p(a) = *priv*, pt(a) = "+", t(a) = *tbl*,
ts(a) < time, g(a) = *rev from AS*;
/* Determine T_u the earliest timestamp of the remaining */
/* authorizations of *user*
to grant authorizations for *priv* on *tbl* */
T_u := min ({ts(a) | $a \in AS$, s(a) = *user*, p(a) = *priv*,
pt(a) = "+", t(a) = *tbl*, go = "yes"}, r_tm)
/* Delete all the negative authorizations for *priv* on *tbl* */
/* granted by *user* before *time* */
delete all authorizations a such that p(a) = *priv*,
pt(a) = "−", t(a) = *tbl*,
ts(a) < T_u, g(a) = *user from AS*
/* Ask for revocation of the authorizations granted by *user* */
/* before time T_u */
for each s_i such that exists a $\in AS$, s(a) = s_i
p(a) = *priv*, t(a) = *tbl*, g(a) = *user*,
ts(a) < T_u **do**
casc-revoke(s_i, *priv, tbl, user*, T_u)
endfor
end

Figure 8: Revised cascading revoke algorithm

changed in an analogous way. The revocation without cascade of privilege p from user y by user x has to: (i) specify with x as grantor all the authorizations in AS granted by y using the grant option given to him by x, (ii) eliminate from the authorization state all the authorizations which would have not existed if y had never received the privilege from x. Note, again, that this corresponds to a revocation with cascade on the authorization state produced after the addition of the new authorizations with s as grantor. An algorithm implementing the revocation of privileges without cascade extended to the consideration of negative authorizations is illustrated in Figure 9.

To illustrate an example of revocation of privileges with the consideration of negative authorizations, consider the authorization state of Figure 7(b) and suppose that user A revokes the select privilege on table T from user B. Figure 10(a) illustrates the authorization states resulting in case of cascade revocation. Figure 10(b) illustrates the authorization states resulting in case of non cascade revocation.

Algorithm 4
revoke(revokee, privilege, table, revoker)

begin
r_tm := current-time
/* Determine T_r earliest time at which *revoker* granted */
/* *privilege* on *table* with the grant option to *revokee* */
T_r := min ({ts(a) | $a \in AS$, s(a) = *revoker*,
p(a) = *privilege*, pt(a) = "+",
t(a) = *table*, g(a) = *revoker*, go(a) = "yes"})
/* Delete all the authorizations for *privilege* on *table* */
/* granted by *revoker* to *revokee* */
delete all authorizations a such that s(a) = *revokee*,
p(a) = *privilege*, pt(a) = "+",
t(a) = *table*, g(a) = *revoker from AS*
/* Determine T_u the earliest timestamp of the remaining */
/* authorizations of *revokee* for p on t with the grant option */
T_u := min ({ts(a) | $a \in AS$, s(a) = *revokee*,
p(a) = *privilege*, pt(a) = "+",
t(a) = *table*, go(a) = "yes"}, r_tm)
for each a $\in AS$, p(a) = *privilege*, t(a) = *table*,
g(a) = *revoker* **do**
if ts(a) > T_r, $s(a) \neq$ *revokee* and $s(a) \neq$ *revoker*
/* the authorization was granted after receiving */
/* the authorization from *revoker* */
then *add* ⟨s(a),p(a),t(a),ts(a),x,go(a)⟩ *to AS*
endif
if ts(a) < T_u /* *revokee* cannot appear as grantor */
/* of the authorization any more */
then *delete a from AS*
endif
end

Figure 9: Revised noncascading revoke algorithm

5 Correctness of algorithms

In this section we prove the correctness of our model. To state what correctness means we need to recall the definition of authorization chain introduced by Fagin [5].

An authorization chain is a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of authorizations such that:

- the grantor of a_1 is the creator of the table: $g(a_1) = \text{owner}(t(a_1))$
- the timestamp of an authorization is bigger that the timestamp of the authorization preceding it in the chain: $\forall i, i = 2, \ldots, n : t(a_i) > t(a_{i-1})$
- the grantor of each authorization is the grantee of the authorization preceding it in the chain: $\forall i, i = 2, \ldots, n : g(a_i) = s(a_{i-1})$
- all authorizations except possibly the last one are with the grant option: $\forall i, i = 1, \ldots, n-1 : go(a_i) =$ "yes"

An authorization a is valid iff it was granted there exists an authorization chain $\langle a_1, a_2, \ldots, a_n \rangle$ in AS

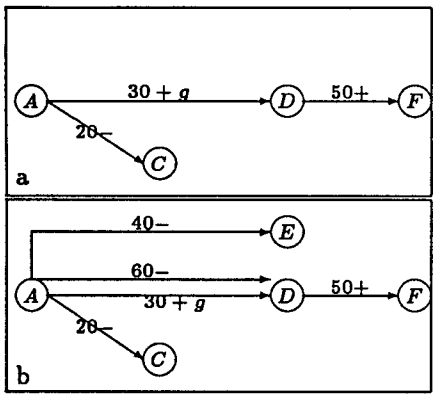

Figure 10: Example of revoke operations when negative authorizations are present

with $a_n = a$ whether all authorizations $a_1, \ldots, a_n$ are unrevoked, i.e., their grantors have not required for their revocation. The consideration of the revocation without cascade require to consider as granted also the authorizations whose grant was not required by the users but which have to be considered as granted as required by the semantics of the non cascade revocation.

An authorization state is correct if it contains all authorizations that are valid.

The correctness of the authorization state after a grant request and after a revocation of a negation is trivial. We now prove that the authorization state is correct after revocation of privileges. Theorem 1 and Theorem 2 prove that the algorithms for the revocation with and without cascade preserve the correctness of the authorization state, i.e., applied to a correct authorization state produce a correct authorization state, reflecting the revoke operation.

In the following we will denote with VALID the set of valid authorizations, and with REV the set of authorizations whose revocation is explicitly required by users.

Theorem 1 *Let AS be an authorization state and $\langle y, p, t, x\rangle$ be a request for revocation with cascade of privilege p on table t from user y by user x. Authorization state AS' resulting from the application of Algorithm* **cascade-revoke** *is correct.*

PROOF We prove that VALID $= AS'$ by proving that $a \in VALID \Leftrightarrow a \in AS'$

Let us first prove implication $a \in$ VALID $\Rightarrow a \in AS'$, i.e., no valid authorization has been deleted. Let us suppose not and derive a contradiction. Suppose then that some valid authorizations have been deleted in the revocation process. Let a be the authorization among them with the minimum timestamp. Therefore $a \in$ VALID and $a \notin AS'$. Since a is valid, there exists a chain of authorizations $\langle a_1, \ldots a_n, a\rangle$ in AS, with $a_1, \ldots a_n, a \notin$ REV. Since a is valid all authorizations $a_1, \ldots, a_n$ are also valid. Hence, since a is the valid authorization with the minimum timestamp that has been deleted, all other authorizations preceding a in the chain have not been deleted in the revocation. Therefore $a_1, \ldots, a_n \in AS'$. However, since a was deleted, either $a \in$ REV, or a was deleted in the recursion. By assumption, since a is valid, it cannot be $a \in$ REV, therefore, a must have been deleted in the recursion. Since $a_n \in AS'$, and a_n directly precedes a in an authorization chain, the minimum timestamp T_u of the authorizations of user $g(a)$ to grant p on t cannot be bigger than $ts(a_n)$, i.e, $T_u \leq ts(a_n) < ts(a)$. Then, since *casc-revoke* for $g(a)$ was called with *time* $= T_u$, authorization a could not have been deleted, and we have a contradiction.

Let us now prove implication $a \in AS' \Rightarrow a \in$ VALID, i.e., AS' contains only valid authorizations. Since no authorization is added by the revocation algorithm we need only to show that all non valid authorizations have been deleted. Suppose not. Let a be the non valid authorization in AS' with the minimum timestamp. Since all authorizations in REV have been deleted, $a \notin$ REV. Since a is not valid, there does not exist an authorization chain $\langle a_1, \ldots a_n, a\rangle$ in AS, with $a_1, \ldots a_n, a \notin$ REV. However since AS was correct by hypothesis, there existed such a chain in AS. Since a is not valid, neither is a_n. Then, since a is the non valid authorization in AS' with the minimum timestamp, $a_n \notin AS'$. Then, when the last a_n preceding a in a chain was deleted by the algorithm the minimum timestamp of the remaining authorizations for $s(a_n) = g(a)$ to grant p on t was bigger that $ts(a)$. Then, $T_u > ts(a)$ and a was deleted, which contradicts the assumption.

□

Theorem 2 *Let AS be an authorization state, and $\langle y, p, t, x\rangle$ be a request for revocation of privilege p on table t from user y by user x. Authorization state AS' resulting from the application of Algorithm* **revoke** *is correct.*

PROOF We now prove that $AS' =$ VALID by proving that $a \in$ VALID $\Leftrightarrow a \in AS'$

We first prove the implication $a \in$ VALID $\Rightarrow a \in AS'$. Suppose the implication does not hold and derive a contradiction. Suppose some valid authorizations do not belong to the authorization state resulting from the application of the algorithm. Let a be the authorization among them with the minimum timestamp. Since $a \notin AS'$ either a should have been added and it was not or a was in AS and was incorrectly deleted. Suppose a is an authorization which should have been added and it was not. Since a should have been added, $a = \langle s(a_i), p(a_i), pt(a_i), t(a_i), ts(a_i), x, go(a_i)\rangle$ for some $a_i \in AS$ such that $g(a_i) = y, p(a_i) = p$, and $t(a_i) = t$ with $a_j \in$ REV, i.e., such that there exists a chain $\langle a_1, \ldots a_j, a_i\rangle$ in AS. Therefore, from the definition of authorization chain and from how T_r has been determined, $T_r \leq ts(a_j) < ts(a_i)$, and a was inserted by the algorithm, which contradicts the assumption. Consider now the case where a was incorrectly deleted in the revocation process. Since a is valid, there exists

a chain $\langle a_1, \ldots a_n, a\rangle$ in $\hat{AS}$, with $a_1, \ldots a_n \notin$ REV, where $\hat{AS}$ indicates the authorization state resulting from adding to AS the new authorizations wioth x as grantor as required by the semantics of the revocation. Since a is valid all authorizations $a_1, \ldots, a_n$ are also valid. Hence, since a is the valid authorization with the minimum timestamp that has been deleted, all other authorizations preceding a in the chain have not been deleted. Therefore $a_1, \ldots, a_n \in AS'$. However, since a has been deleted, either $a \in$ REV, or $g(a) = y$ and $ts(a) < T_u$. By assumption, since a is valid, it cannot be $a \in$ REV. Then, $g(a) = y$ and $ts(a) < T_u$. Since $a_n \in AS'$, and there exists an authorization chain such that a_n directly precedes a, the minimum timestamp T_u of the remaining authorizations of y to grant p on t cannot be bigger than $ts(a_n)$. Then, $T_u \leq ts(a_n) < ts(a)$ and a could not have been deleted, which contradicts the assumption.

Let us now prove implication $a \in AS' \Rightarrow a \in$VALID, i.e., AS' contains only valid authorizations. Suppose not and let us derive a contradiction. Let a be the non valid authorization in AS' with the minimum timestamp. Since $a \in AS'$ either a was incorrectly added or has not been deleted.

Let us first suppose that the authorization was incorrectly added. Since a was added, $a = \langle\ s(a_i), p(a_i), pt(a_i), t(a_i), ts(a_i), x, go(a_i)\ \rangle$ such that there exists $a_i \in AS$, $ts(a_i) > T_r$, and hence there existed a chain $\langle a_1, \ldots a_j, a_i\rangle$ in AS with $a_j \in$ REV. Since $a \notin$ VALID, either $a \in$ REV or there do not exist any chain $\langle a_1, \ldots a_n, a\rangle$ in $\hat{AS}$, with $a_1, \ldots a_n, a \notin$ REV. Suppose $a \in$ REV, then $s(a_i) = y$, and a was not added, which contradicts the assumption. Therefore $a \notin$ REV. However, since AS was correct and, by definition, $AS \subseteq \hat{AS}$, there exists a chain $\langle a_1, \ldots a_n, a_i\rangle$ in $\hat{AS}$. Then, from how a was obtained, $\langle a_1, \ldots a_{n-1}, a\rangle$ is also a chain in $\hat{AS}$. Therefore, since $a \notin$ VALID, $a_m \in$ REV for some a_m in the chain. Since a_{n-1} precedes a_n, in the chain and $a_n \in$ REV, $s(a_{n-1}) = x$. Therefore, $a_{n-1} \notin$ REV. Then $a_m \neq a_{n-1}$. Let $a'_{m+1} = \langle\ s(a_{m+1}), p(a_{m+1}), pt(a_{m+1}), t(a_{m+1}), ts(a_{m+1}), x, go(a_{m+1})\ \rangle \in \hat{AS}$ be the new authorization inserted with x as grantor. Since a_{m+1} was added by the algorithm, $s(a'_{m+1}) \neq y$, $a_{m+1} \notin$ REV. Then, $\langle a_1, \ldots a_{m-1}, a'_{m+1}, \ldots, a_{n-1}, a\rangle$ is a chain in $\hat{AS}$, with $a_1, \ldots a_{n-1} \notin$ REV. Hence, a is valid and we have a contradiction.

Let us then suppose that a was not deleted. Since all authorizations in REV have been deleted, $a \notin$ REV. Since AS is correct, there exists $\langle a_1, \ldots a_n, a\rangle$ in AS. Then, since a is not valid, and since $AS \subseteq \hat{AS}$, $a_i \in$ REV for some a_i in the chain. Let us suppose than $a_i \neq a_n$. Since $a_i \in$ REV, authorization $a'_{i+1} = \langle s(a_{i+1}), p, t, ts(a_{i+1}), x, go(a_{i+1})\rangle \in \hat{AS}$, and hence $\langle a_1, \ldots, a_{i-1}, a'_{i+1}, \ldots a_n, a\rangle$ is a chain in $\hat{AS}$, with $a_1, \ldots a_n \notin$ REV. Therefore a is valid and we have a contradiction. If more than one authorization a_i belongs to REV, $a_i \neq a_n$, the reasoning can be applied more than once. Let us then suppose that $a_n \in$ REV. Since $a_n \in$ REV, $s(a_n) = g(a) = y$. Moreover, since a has not been deleted by the algorithm, $ts(a) \geq T_u$, which implies that there existed $a_k \in AS, a_k \notin$ REV such that a_k preceded a in a chain. Therefore, from the reasoning above a is valid and we have a contradiction. □

6 Conclusion and open problems

In this paper, we have proposed two extensions to the System R authorization model for relational databases [5, 6]. The first extension concerns the noncascading revoke operation. Although the recent draft of SQL standard [7] recognizes its need, many details related to its application have been left unspecified. The second extension concerns the negative authorization. Its need is specified for high assurance systems in [4].

There are several open problems that are under investigation by us. We are working on extending our model to incorporate views [6] and groups [11]. In our model, we have assumed that negative authorizations always override positive authorizations. There are other policies which could be applied to decide whether a request of a user to access a table should be granted. We are investigating some of these alternatives.

References

[1] M.M. ASTRAHAN ET AL., "System R: A relational approach to data base management," *ACM TODS*, Vol. 1, No. 2, June 1976, pp. 352–359.

[2] E. BERTINO AND L. M. HAAS, "Views and security in distributed database management systems," *Proc. First International Conference on Extending Database Technology (EDBT)*, Venice (Italy), Springer-Verlag Lecture Notes in Computer Science, Vol. 303, 1988, pp. 155–169.

[3] D. D. CHAMBERLIN ET AL., "A history and evaluation of System R," *Comm. ACM*, Vol. 24, No. 10, 1981, pp. 632–646.

[4] DEPARTMENT OF DEFENSE, *Trusted computer system evaluation criteria*, DoD 5200.28-STD, Dec. 1985

[5] FAGIN, R., "On an authorization mechanism," *ACM TODS*, Vol. 3, No. 3, Sept. 1978, pp. 310-319

[6] GRIFFITHS, P.G., AND WADE, B., "An authorization mechanism for a relational database system," *ACM TODS*, Vol. 1, No. 3, Sept. 1976, pp. 242-255

[7] MELTON, JIM, ED., ANSI X3H2-90-309, "(ISO/ANSI working draft) Database Language SQL2," August 1990.

[8] F. RABITTI, E. BERTINO, W. KIM, AND D. WOELK, "A model of authorization for next-generation database systems," *ACM TODS*, Vol. 16, No. 1, March 1991, pp. 88–131.

[9] T. F. LUNT ET AL., "The seaview security model." *IEEE Transactions on Software Engineering*, Vol. 16, No. 6, June 1990, pages 593–607.

[10] SELINGER, P.G., "Authorizations and views," in *Distributed Data Bases*, I.W. Draffan and F. Pook eds., Cambridge University Press, Cambridge 1980

[11] P. F. WILMS AND B. G. LINSDAY, "A database authorization mechanism supporting individual and group authorization," in *Distributed Database Systems*, R.P. van de Riet and W. Litwin, eds., North-Holland, 1982, pp. 273–292.