FLEXIBLE SECURITY POLICIES IN SQL

Steve Barker

*Cavendish School of Computer Science,
University of Westminster, London, UK*
barkers@westminster.ac.uk

Arnon Rosenthal

*The MITRE Corporation,
202 Burlington Road, Bedford, MA, USA*
arnie@mitre.org

Abstract We show how a wide variety of role-based access control policies may be formally specified in the stratified subset of clause form logic. We then show how these formal specifications may be automatically translated into a small subset of SQL to be used to seamlessly protect an SQL database from unauthorized read and update requests made by authenticated users. We demonstrate the power of our approach by showing how a variety of access control policies can be represented.

Keywords: Role-Based Access Control, Stratified Logic, SQL.

Introduction

In the SQL2 standard [6], the security-specific language features are restricted to simple GRANT and REVOKE statements (albeit views may also be used to provide security). Unfortunately, the GRANT-REVOKE model enables a *security administrator (SA)* to represent only a small subset of the access control policies that are needed in practice.

While RDBMS products and the SQL3 standard include some support for roles as well as GRANT and REVOKE, there are still many important policies that existing products and SQL cannot express. To get more expressive power, application programs are typically used. However, this approach complicates the maintenance of an access control policy and makes it difficult for SAs to reason about their behavior. We therefore envisage making use of a high-level

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35587-0_24](https://doi.org/10.1007/978-0-387-35587-0_24)

M. S. Olivier et al. (eds.), *Database and Application Security XV*

© IFIP International Federation for Information Processing 2002

policy specification language which enables a range of access control policies to be formulated, and which can be translated into an implementation of the policy using SQL views and triggers. Furthermore, we aim for the translation from specification to implementation to be automated.

In our proposal, function-free, *stratified clause logic* [10] is used by a SA to formulate RBAC policies that define the set of authorized actions that can be performed on a database.* We believe that most realistic access control policies may be expressed in the high-level language of stratified logic by using a small number of clauses which have a simple declarative and operational meaning that makes it relatively easy for SAs to reason about the consequences of a specification of an access policy. Representing access policies directly in SQL and reasoning about their effects is much harder. Stratified logic and SQL share a well-defined and uncontroversial model-theoretic semantics (the *perfect model semantics* [12]), stratified logic has a simple equivalence preserving translation into SQL [1], and this translation may be automated.

There are certainly other computational options to the one we propose, e.g., using a logic engine, a hybrid of SQL and logic, and even implementing high performance parts in C++. However, there are several reasons why the translation approach is worth investigating. It is natural to consider managing the authorization system for an SQL database by using SQL alone, it is attractive to execute only SQL, instead of adding a logic execution engine, it is attractive to provide security and transaction protection using the ordinary RDBMS facilities, and by avoiding extra software systems (beyond the translator), we avoid requiring experts to manage and maintain them. Moreover, there are a number of attractive technical results which are applicable to our approach.

The RBAC models that we specify in stratified logic are based on the family of models described in [14]. That is, we assume that an RBAC theory includes role and permission assignments ($RBAC_0$), and may additionally include specifications of either role hierarchies ($RBAC_1$) or constraints ($RBAC_2$). $RBAC_3$ includes role hierarchies and constraints.

The rest of this paper is organized thus. In Section 1, we provide a basic introduction to stratified logic, and we discuss some relevant notions from database and access control theory. In Section 2, we describe the formulation of a generic $RBAC_1$ model as a stratified theory, an $RBAC_1$ theory. In Section 3, we show how an $RBAC_1$ theory can be translated into SQL and used to control access to an SQL database. In Section 4, we demonstrate the power of our approach by showing how a range of access control policies may be represented in a small subset of SQL. In Section 5, we further extend our representation of $RBAC_1$ to include constraints on $RBAC_3$ theories, and we consider constraint

* A GUI may be used to help a SA formulate policies expressed using stratified clauses.

enforcement. In Section 6, some conclusions are drawn and suggestions for further work are made.

Henceforth, any references to SQL and databases should be, respectively, interpreted as referring to SQL2 (plus a WITH RECURSIVE construct and triggers) and SQL databases. In the sequel, we use upper case **bold type** for indicating reserved words of SQL and lower case *italic type* for user-supplied information in SQL statements.

1. PRELIMINARIES

In our proposal, RBAC theories are formally specified in a subset of function-free normal clause logic. A normal clause is an expression of the form: $H \leftarrow L1, L2, \dots, Lm$ ($m \geq 0$). The *head* of the clause, H , is an *atom* and $L1, L2, \dots, Lm$ is a conjunction of *literals* that constitutes the *body* of the clause. If $L1, L2, \dots, Lm$ is true (proved) then H is true (proved). A literal is an atomic formula or its negation; in this paper negation is *negation by failure* [16], and the negation of the atom A is denoted by *not A*. A *normal theory* is a finite set of normal clauses. A clause with an empty body is an *assertion* or a *fact*. A clause with an empty head is a *denial*. A *definite clause* is a normal clause with no negative literal in its body; a *definite theory* is a finite set of definite clauses.

In the clause $H \leftarrow L1, L2, \dots, Lm$, H is said to positively (negatively) *refer* to the literal Li ($i \in \{1, \dots, m\}$) if Li is a positive (negative) literal in the definition of H . A *dependency graph* [16] has an edge labeled $+$ ($-$) if H refers to Li positively (negatively). A normal theory is *stratified* [16] iff its dependency graph contains no cycle with a negative edge. A stratified theory is a finite set of stratified clauses.

In this paper, the constants that appear in clauses are denoted by lower case symbols; variables are denoted by using symbols that appear in the upper case.

In the sequel we will relate the discussion of stratified logic and SQL to subsets of *SPCUD-algebra* [1]. The SPCUD-algebra is a language that includes the five primitive operators of relational algebra viz.: select, project, cartesian product, union and set difference. The *SPCU-algebra* is SPCUD-algebra without the set difference operator; *SPC-algebra* is SPCU-algebra without the union operator.

Since the RBAC theories that we consider are stratified, these theories have a unique *perfect model* [12]. The perfect model of an RBAC theory is the set of logical consequences of the theory and includes the set of authorized accesses defined by the theory. These authorizations will be defined by a predicate *permitted*(U, P, O) where U denotes a user of a database, P denotes a permission and O is an object. The permissions we consider include insert, delete, update and select (read). The objects to be protected are base tables, views and attributes in base tables.

Given an RBAC theory Σ and a request by U to perform a P operation on O , the approach adopted for access control involves deciding whether an instance of $permitted(U,P,O)$ is authorized by Σ . Informally, the desired enforcement semantics are:

IF permitted(U,P,O) THEN execute the P action on O for U ELSE provide informative error message and abort

As we will see, the approach we adopt for evaluating access requests is based on *query modification* [15]. That is, security information is used to modify a user's query such that the revised form of the query satisfies access control restrictions.

2. REPRESENTING $RBAC_1$ AS A STRATIFIED THEORY

Any RBAC model provides means for creating associations between a role and (1) permissions to perform operations on database objects and (2) a set of users. These may be represented in clause form logic by defining $ura(U,R)$ and $rpa(R,P,O)$ predicates (see [3] for details).

In the $ura(U,R)$ relation, the predicate name ura is shorthand for *user-role assignment*; definitions of ura are used to represent that user U is assigned to the role R . Similarly, $rpa(R,P,O)$ stands for *role-permission assignment*; definitions of rpa are used to specify that role R is assigned the P permission on the object O .

An instance of an $RBAC_1$ role hierarchy may be represented using a set of $d-s$ facts (where $d-s$ is short for "directly senior to"); $d-s$ is an irreflexive, intransitive and non-symmetric relation. The assertion $d-s(r_i, r_j)$ records that the role r_i is directly senior to the role r_j in an $RBAC_1$ role hierarchy.

The clauses that follow define a *senior_to* relation as the reflexive-transitive closure of the $d-s$ relation (where ' $_$ ' is an anonymous variable):

```
senior_to( $R1, R1$ )  $\leftarrow d-s(R1, \_)$ 
senior_to( $R1, R1$ )  $\leftarrow d-s(\_, R1)$ 
senior_to( $R1, R2$ )  $\leftarrow d-s(R1, R2)$ 
senior_to( $R1, R2$ )  $\leftarrow d-s(R1, R3), senior\_to(R3, R2)$ 
```

To define authorized access, the following clause may be used:

```
permitted( $U, P, O$ )  $\leftarrow ura(U, R1), senior\_to(R1, R2), rpa(R2, P, O)$ 
```

The *permitted* clause expresses that a user U has the P permission on an object O if U is assigned to a role $R1$ that is senior to a role $R2$ in an $RBAC_1$ role hierarchy and $R2$ is assigned the P permission on O . That is, U has the P permission on O if U is assigned to a role that inherits the P permission on O via an $RBAC_1$ role hierarchy.

The definition of *permitted* implies that a *closed access policy* [5] is to be used to protect a database. That is, a user's permission to access information must be specifically authorized in an access control theory; users do not have access to information in the absence of a denial prohibiting the access (*an open policy* [5]). However, any number of access policies can be represented in our approach by making simple modifications to the definition of *permitted* and the specification of permissions.

To see how different policies may be represented, suppose that an open policy is required such that a user U has the P permission on an object O if U is not explicitly denied this permission. For this policy, definitions of a $d-rpa(R,P,O)$ predicate are included in an $RBAC_1^-$ theory instead of rpa . Here, $d-rpa$ is short for "denied role permission assignment" and $RBAC_1^-$ is $RBAC_1$ with denials of access expressed via $d-rpa$ predicates. The $d-rpa(R,P,O)$ clauses in an $RBAC_1^-$ theory are used to specify that role R is denied the P permission on O . The required open access policy may be represented by the following (stratified) clause:

$$permitted(U,P,O) \leftarrow not\ denied(U,P,O)$$

The definition of the auxiliary *denied* relation in $RBAC_1^-$ is as follows:[†]

$$denied(U,P,O) \leftarrow ura(U,R1), senior_to(R2,R1), d-rpa(R2,P,O)$$

The definition of *denied* specifies that negative permissions are inherited downwards in an $RBAC_1^-$ role hierarchy (an $RBAC_1^-$ role hierarchy is as defined in $RBAC_1$). That is, if U is assigned to the role $R1$, the role $R2$ is senior to $R1$ in an $RBAC_1$ role hierarchy, and $R2$ is denied the P permission on O then $R1$ and hence U is denied P access on O . An instance of $permitted(U,P,O)$ will hold in an $RBAC_1^-$ theory iff there is no instance of $denied(U,P,O)$ that overrides the default assumption that $permitted(U,P,O)$ holds.

Any number of hybrid (open/closed) policies may be expressed using stratified clauses, and open and closed policies may be specified as coexisting in a single theory (with *various conflict resolution policies* [9] being represented to

[†]This definition assumes that every user is assigned to at least one role and that there is a single $RBAC_1$ role hierarchy defined in an $RBAC_1$ theory.

manage situations where an instance of a (U, P, O) triple is both permitted and denied).

Apart from the clauses that define *permitted* (or *denied*) all other clauses in an RBAC theory are application-specific. The fact that a small number of application non-specific clauses are required to express RBAC policies is important since it makes the representation of a policy, reasoning about the policy and the maintenance of the policy relatively simple. What is more, these clauses may be equivalently represented by using only a small number of SQL statements.

In RBAC, users are usually required to be active in appropriate roles when making access requests. In our representation of $RBAC_1$ as a stratified theory, *sessions* [14] may be modeled by using an $activate(U, R1)$ predicate. A ground instance of $activate(U, R1)$ is appended to the $RBAC_1$ theory if user U successfully activates role $R1$. Conversely, a ground instance of $activate(U, R1)$ is removed from an $RBAC_1$ theory if U deactivates role $R1$. If $activate$ predicates are included in an $RBAC_1$ theory then the definition of *permitted* is extended to include an $activate(U, R1)$ condition to ensure that the user U is active in the role $R1$ at the time at which U makes an access request.

3. $RBAC_1$ IN SQL

We represent user-role and permission-role assignments as logic predicates that have a one-to-one correspondence with SQL tables. There is a table for d -s and tables for the explicit assignments of users and permissions to roles. Intensional predicates define implicit accesses; these are translated into SQL views.

Since *permitted*, *senior_to* and d -s are specified using definite clauses and *ura* may reasonably be assumed to be defined by a set of ground assertions, it follows that if the definitions of *rpa* clauses in an $RBAC_1$ theory are expressed in definite clause logic then the $RBAC_1$ theory will be definite. It is well known that function-free, non-recursive definite clause form theories (i.e., non-recursive Datalog theories [16]) can be translated into an equivalent set of statements in *SPCU-algebra* [1]. Since SQL is sufficiently powerful to represent any expression in a *relationally complete* language [6], it follows that SQL is sufficiently powerful to enable any formula in *SPCU-algebra* to be expressed. From [1], *SPC-algebra* expressions can be equivalently represented in SQL using the **SELECT-FROM-WHERE** construct, and the **UNION** operator of SQL corresponds to the union operator of relational algebra. For representing the recursive definition of *senior_to* in SQL, the **WITH RECURSIVE** construct is sufficient since *senior_to* is defined by a set of *linear recursive* [2] and definite clauses. If *safe* [16], function-free and recursion-free stratified clauses are used to represent the *ura* and *rpa* clauses that appear in an $RBAC_1$ theory then the

theory can be equivalently represented in *SPCUD*-algebra [1]. To represent the n negative literals that may appear in clauses in an $RBAC_1$ theory, SQL statements that include **NOT EXISTS** (or **NOT IN**) may be used [1].

3.1. $RBAC_1$ in SQL: Representational Issues

In this section we discuss the representation of $RBAC_1$ theories in SQL. Since we assume that *ura* is a set of ground atomic assertions, the set can be represented by a table. To represent, in SQL, a clause form definition of *rpa* with a non-empty set of literals in its body that protects a relation r , a **CREATE VIEW** definition is used to define the subset of r that members of a given role are permitted to access. A **CREATE TABLE** *rpa* statement is used to record the set of permissions users have on the views and base tables in the database. In the case of the binary *ura* relation, we will assume that the attribute names used in the **CREATE TABLE** statement are U and R , for user and role respectively. For the ternary relation *rpa*, we assume that the attribute names are R , P and O where R is for role, P is for permission and O is for object. A **CREATE TABLE** statement is also required to create a 2-attribute base table corresponding to the *d-s* relation.

A special role *public* may be used to specify public access on objects. For that, we may include a *d-s*(r_i ,*public*) assertion in an $RBAC_1$ theory for each role $r_i \in Role(X)$ (where $Role(X)$ is the set of roles appearing in an $RBAC_1$ theory) such that $\neg \exists r_j [r_i \rightarrow r_j]$ holds in an $RBAC_1$ role hierarchy (where $r_j \in Role(X)$). The *public* role is the unique least element in the partial order of roles that comprises an $RBAC_1$ role hierarchy (i.e., $\forall r_k \in Role(X)$ *senior_to*(r_k ,*public*)). Appropriate instances of *rpa*(*public*, P , O) are added to the access policy specification to record that the P permission on O is assigned to *public*.

The *senior_to* relation may be defined in terms of **WITH RECURSIVE**. In the ensuing discussion we will assume that *senior_to* includes the pair of attributes (*senior*,*junior*). Each tuple (r_i , r_j) in *senior_to* records that role r_i is senior to role r_j in an $RBAC_1$ role hierarchy.

Finally, to represent *permitted*, an SQL view is used to define the set of (U , P , O) triples such that the user U has the P permission on O . The required SQL is:

```
CREATE VIEW permitted( $U$ , $P$ , $O$ ) AS
SELECT ura. $U$ , rpa. $P$ , rpa. $O$ 
FROM ura, rpa, senior_to
WHERE ura. $R$ =senior_to.senior AND senior_to.junior=rpa. $R$ ;
```

To implement our approach, *permitted* could be stored as a *materialized view* [1] that is recomputed (in an incremental way) whenever changes are made to the relations in terms of which *permitted* is defined. Similarly, *senior_to* is stored as a materialized relation to avoid its recomputation each time an access request is evaluated; *senior_to* is recomputed each time the set of *d-s* assertions is modified.

To represent *permitted* in SQL when *activate* assertions are included in an $RBAC_1$ theory, a **CREATE TABLE** statement is required to record the set of all pairs (U, RI) such that the user U has active the role RI . The only modification required to the view definition of *permitted* in SQL is for the following conjunctive condition to be added to the **WHERE** clause: **AND** *ura.U=activate.U* **AND** *ura.R=activate.R*.

3.2. $RBAC_1$ in SQL: Evaluating Access Requests

To ensure that only authorized access to an SQL database is allowed, **EXISTS** subqueries, which include security information, are added to the access requests a user U makes. For example, if u is the identifier for an authenticated user requesting to retrieve all information in a table $t1$ (for which read permission is required) then the required SQL is:

```
SELECT *
FROM t1
WHERE EXISTS
(SELECT * FROM permitted WHERE permitted.U='u'
AND permitted.P='read' AND permitted.O='t1');
```

The above approach has the elegance of existing entirely within the representation of $RBAC_1$ as a stratified theory. By going beyond what is representable as a single logic clause, we can implement the desired semantics of Section 2 viz.:

IF (*appended predicate*) **THEN** (*query*) **ELSE** (*error message*)

The **IF-THEN** form has two pragmatic advantages. First, it lets users distinguish between denial and empty result; second, it will be efficient even with query processors that do not handle nested predicates efficiently. With either approach, we envisage security management tools being used to modify the request before submitting it to a DBMS. Alternatively, a DBMS would be extended to handle this modification itself.

The ' u ' value in the *permitted.U* condition is taken from user u 's login details which are provided as part of the authentication process; the *read* value in

permitted. P = 'read' comes from the fact that a request to SELECT is made by u ; and the $t1$ value in the *permitted*. O condition is taken from the FROM clause in the SQL SELECT statement that u issues. A *permitted*. O = t_i (*permitted*. O = v_i) condition is required for each table t_i (view v_i) that appears in the FROM clause of a SELECT statement and which is read protected.

Example 1 Consider the following relations from [6]: $S(\underline{Sno}, Sname, Status, City)$ for Suppliers, $P(\underline{Pno}, Pname, Color, Weight, City)$ for Parts, $J(\underline{Jno}, Jname, City)$ for Projects, and $SPJQ(\underline{Pno}, \underline{Sno}, \underline{Jno}, Quantity)$ (recording the quantity of parts suppliers supply to each project). Primary keys are underlined and Pno , Sno and Jno are foreign key attributes in $SPJQ$. Suppose that a query, to list all project numbers for projects supplied with red parts by suppliers located in London, is posed by the authenticated user Sue on a database, Θ , that includes the S , P , J and $SPJQ$ relations (all of which are read protected). The secure form of the SQL query on Θ is:

```
SELECT JNo
FROM SPJQ, S, P, J, permitted
WHERE J.JNo=SPJ.JNo AND SPJQ.PNo=P.PNo AND
P.Colour='red' AND SPJQ.SNo=S.SNo AND S.City='London' AND
EXISTS(SELECT * FROM permitted WHERE permitted.U='Sue'
AND permitted.P='read' AND permitted.O='SPJQ') AND
EXISTS(SELECT * FROM permitted WHERE permitted.U='Sue'
AND permitted.P='read' AND permitted.O='S') AND
EXISTS(SELECT * FROM permitted WHERE permitted.U='Sue'
AND permitted.P='read' AND permitted.O='P') AND
EXISTS(SELECT * FROM permitted WHERE permitted.U='Sue'
AND permitted.P='read' AND permitted.O='J');
```

That is, a JNo value may be disclosed to Sue iff it is true in Θ that a project with the JNo is supplied with red parts by a supplier located in London, and Sue is authorized to read the $SPJQ$, S , J and P relations that are required to be accessed in order for her to know that the JNo value is true in Θ .

◇

For protecting SQL databases from unauthorized modifications the same approach as that described above is used.

Example 2 Suppose that Sue wishes to change the color of all red parts to yellow. For that, the secure SQL is:

```
UPDATE P
SET Color='yellow'
WHERE Color='red' AND
```

**EXISTS(SELECT * FROM *permitted* WHERE *permitted.O*='P'
AND *permitted.P*='update' AND *permitted.U*='Sue');**

◇

In the examples of access control that we have given, the objects that have been referred to are base tables and views. However, it is also possible to protect the individual attributes appearing in a table. For example, a tuple (*r1, read, t.a*) may be included in *rpa* to record that role *r1* is assigned the read permission on the attribute *a* in the table *t*. The access control information on attributes may be used to rewrite user access requests in a secure way by using the approach we have described.

4. BEYOND *RBAC*₁

The *RBAC*₁ policy that we have described thus far is a simple one and could be represented in some existing RDBMSs. However, an attraction of our approach is that it can be extended to implement various access policies that cannot be represented in existing RDBMSs, and without requiring any language features beyond those included in SQL. Since access requirements are often application and organization-specific, the flexibility that our approach offers is important in practice.

To demonstrate the power of our approach, we consider the representation of a hybrid access control policy which allows positive permissions to be expressed using *rpa* definitions and denials to be expressed via *d-rpa* definitions. The denials may be used to override positive permissions (i.e., a *denials take precedence* conflict resolution policy is enforced [9]). The set of authorized accesses may be represented in stratified logic thus (where *permitted* and *denied* are as defined in Section 3):

access(U,P,O) ← permitted(U,P,O), not denied(U,P,O)

The *access* clause specifies that *U* has the *P* permission on *O* if *U* has been assigned the *P* permission on *O* and this (positive) permission is not overridden by a denial of *P* access on *O* to *U*.

The translation of *access* into SQL produces the following view definition (where *permitted* and *denied* are views defined in SQL):

**CREATE VIEW *access(U,P,O)* AS
SELECT * FROM *permitted*
WHERE NOT EXISTS(SELECT * FROM *denied*);**

The conditional form used in this case would be:

IF NOT *denied* **THEN** (*user query*) **ELSE** (*error message*)

A natural generalization is to allow the assignment of limited permissions. Instead of simply assigning a permission to a role (a pair, easily represented by a table), one may also attach a *limitation predicate* [13] to be evaluated before the permission can be used. In this case, the SQL will include the limitation predicate in the appended condition. Moreover, any number of access policies may be represented by using the approach we have described.[‡]

5. $RBAC_3$ IN SQL

In this section, we consider the representation of constraints and thus $RBAC_3$ policies in stratified logic and SQL. The high-level nature of stratified logic makes it easy for a SA to express constraints (albeit a GUI may be used to help a SA to formulate constraints). Moreover, the translation of constraints expressed in stratified logic into SQL can be easily and efficiently automated.

In our approach, constraints may be expressed in denial form viz:
 $\leftarrow L1, L2, \dots, Ln$ (where Li ($i=(1..n)$) is a literal and $n > 0$). The reading of this statement is that it is impossible for the conjunction of literals $L1 \wedge L2 \wedge \dots \wedge Ln$ to evaluate to true (be satisfied) in a $RBAC_3$ theory. Since denials may be expressed using constants or variables, it is possible for a SA to make the constraints on an $RBAC_3$ theory as specific or as general as is required. However, we believe that constraints should be written in as general a form as is possible, and should be specialized at the time of checking by using application-specific assertions in an $RBAC_3$ theory. These assertions may be represented by using base tables in the SQL representation of an $RBAC_3$ theory, and enable Nicolas' Simplification Method [11] to be exploited when checking constraints.

Constraints expressed in denial form have a very straightforward representation as assertions in SQL. To see that, consider a *static separation of duty* constraint [14] expressed thus:

$\leftarrow ura(U, R1), ura(U, R2), ssd(R1, R2)$

This constraint specifies that: it is impossible for a user to be recorded in an $RBAC_3$ theory as being assigned to a pair $(R1, R2)$ of roles that are specified as being statically separated. A set of *ssd* assertions is used to record specific pairs of roles that are statically separated. To represent the set of *ssd* assertions in SQL, a **CREATE TABLE** *ssd(role T_i , excluded T_j)* statement may be used (where T_i and T_j are the types associated with the attributes *role* and *excluded* (i.e., roles)). The *ssd* table is populated with tuples consisting of a pair of values

[‡]For example, the temporal authorization model described in [4].

(r_i, r_j) (where $r_i, r_j \in \text{Role}(X)$). Because of the symmetry of the *ssd* relation, the *ssd* table will additionally include the pair of roles (r_j, r_i) ($i \neq j$).

Since a denial expresses negation and variables are universally quantified, $\leftarrow L1, L2, \dots, Ln$ may be expressed by the universally closed sentence $\forall \neg (L1 \wedge L2 \wedge \dots \wedge Ln)$ and thus the existentially closed sentence $\neg \exists (L1 \wedge L2 \wedge \dots \wedge Ln)$. Hence, to express the *ssd* constraint in SQL the following statement may be used and is checked each time a tuple is inserted into *ura*:

```
CREATE ASSERTION ssd CHECK
NOT EXISTS
(SELECT subject, ssd.role, ssd.excluded
FROM ura U1, ura U2, ssd
WHERE U1.S=U2.S AND U1.R=ssd.role AND U2.R=ssd.excluded);
```

Unfortunately, the ASSERTION above involves some expensive join conditions. A more efficient approach is to use triggers.[§]

To formally specify triggers, *range form* formulae may be used [8]. A statement is in range form if it is an existentially quantified first order sentence. Constraints expressed in range form have a simple translation into an equivalent SQL form. Equivalence here means that if a change to an *RBAC*₃ theory does not violate a constraint *I* expressed in denial form or its equivalent range form then the change to the *RBAC*₃ theory expressed in SQL is not rejected by the representation of *I* as a trigger in SQL. The translation of constraints in range form into SQL can be easily and efficiently automated. Moreover, it is simple to enforce assertions *incrementally* by triggers whose logic exploits the fact that before the current update the constraint was satisfied.

As is usual in RDBMSs, we assume that two cached and user inaccessible relations *inserted* and *deleted* exist with *n* attributes corresponding to the degree of the table into which a tuple is to be inserted or deleted (in the sequel we will use the notation *r.i* to refer to the *i*-th argument of the tuple to be inserted/deleted from an arbitrary relation *r*). The *inserted* (*deleted*) relation is used to store the tuples to be inserted (deleted) in a change transaction on an *RBAC*₃ theory. The triggers that define the constraints on an *RBAC*₃ theory represented in SQL may be expressed in terms of violations of the constraint by specifying that a ROLLBACK action is to be performed if the constraint is violated.

For the *ssd* constraint, the required trigger is as follows (see [8] for details of the translation of range form into trigger form):

```
CREATE TRIGGER ON ura FOR INSERT
```

[§] Triggers are proposed for inclusion in the SQL3 standard and are already supported in commercial RDBMSs.

```

IF(EXISTS (SELECT * FROM inserted WHERE
EXISTS (SELECT * FROM ura u1, ssd
WHERE u1.user=inserted.1 AND inserted.2=ssd.role) AND
EXISTS (SELECT * FROM ura u2, ssd WHERE ssd.excluded=u2.role)))
ROLLBACK

```

The meaning of this SQL statement is as follows: if (u_i, r_j) is the pair of values in the *ura* tuple that is proposed for insertion into an $RBAC_3$ theory, u_i is also recorded as being assigned to a role r_k in *ura*, and (r_j, r_k) is a tuple in the table *ssd* that records pairs of roles that are statically separated then the insert request is rolled back since otherwise the *ssd* constraint on an $RBAC_3$ theory would be violated.

Any number of other types of (static) constraint on an $RBAC_3$ theory (e.g., cardinality constraints on roles) can be represented in essentially the same way that *ssd* is.

6. CONCLUSIONS AND FURTHER WORK

The great challenge for improving the formulation of access control policies is to provide practitioners with usable, high-level constructs which possess clear semantics. We have shown how that can be done. The approach that we have described enables a SA to specify a variety of access control policies using elements of RBAC that map directly into stratified logic. Moreover, we have demonstrated how these constructs can be implemented by using a small subset of SQL3.

The development of tools for the translation of RBAC theories into SQL is a matter for further work. We also intend to investigate how the proof procedures used in stratified logic may be exploited to help administrators to reason about the access control policies they specify.

References

- [1] Abiteboul, S., Hull, R., and Vianu, V., *Foundations of Databases*, Addison Wesley, 1995.
- [2] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., Magic Sets and Other Strange Ways to Implement Logic Programs, *Proc. ACM PODS*, 1-15, 1986.
- [3] Barker, S., Data Protection by Logic Programming, *1st International Conf. on Computational Logic (CL2000)*, LNAI 1861, 1300-1313, Springer, 2000.
- [4] Barker, S., and Rosenthal, A., Flexible Security Policies in SQL, *Proc. 15th IFIP Conf. on Database Security*, 2001. Also available at <http://www.mitre.org/resources/centers/it/staffpages/arnie/>
- [5] Castano, S., Fugini, M., Martella, G., and Samarati, P., *Database Security*, Addison Wesley, 1995.
- [6] Date, C., *An Introduction to Database Systems*, Addison Wesley, 2000.
- [7] Date, C., and Darwen, A., *A Guide to the SQL Standard*, Addison Wesley, 1993.

- [8] Decker, H., Soundcheck for SQL, *3rd International Symposium on Practical Applications of Declarative Languages (PADL'01)*, LNCS 1990, Springer, 214-228, 2001.
- [9] Jajodia, S., Samarati, P., and Subrahmanian, V., A Logical Language for Expressing Authorizations, *IEEE Symp. on Security and Privacy*, 94-107, 1997.
- [10] Lloyd, J., *Foundations of Logic Programming*, 2nd Ed., Springer, 1987.
- [11] Nicolas, J., Logic for Improving Integrity Checking in Relational Databases, *Acta Informatica*, 18(3), 227-253, 1982.
- [12] Przymusiński, T., Perfect Model Semantics, *Proc. 5th ICLP*, MIT Press, 1081-1096, 1988.
- [13] Rosenthal, A., and Sciore, E., Extending SQL's Grant and Revoke Operations to Limit and Reactivate Privileges, *Proc. 14th IFIP Conf. on Database Security*, 2000.
- [14] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C., Role-Based Access Control Models, *IEEE Computer*, 29(2), 38-47, 1996.
- [15] Stonebraker, M., Implementation of Integrity Constraints and Views by Query Modification, *Proc. SIGMOD Workshop on Management of Data*, 65-78, 1975.
- [16] Ullman, J., *Principles of Database and Knowledge-Base Systems: Volume 1*, Computer Science Press, 1990.