RBAC POLICY IMPLEMENTATION FOR SQL DATABASES

Steve Barker
King's College, U.K.

Paul Douglas
Westminster University, U.K.

Abstract We show how specifications of role-based access control (RBAC) and temporal role-based access control (TRBAC) policies in a logic language may be used in *practical* implementations of access control policies for protecting the information in SQL databases from unauthorized retrieval and update requests. Performance results for an implementation of a variety of RBAC policies for protecting an SQL databases and some optimization methods that may be used in implementations are described.

Keywords: RBAC, SQL, Internet Database.

1. Introduction

The protection of SQL databases from unauthorized access requests has long been recognized as being important. However, SQL standards and SQL products have hitherto included only limited options for expressing the protection requirements for the information contained in SQL databases.

In the SQL2 standard [11], the security-specific language features are restricted to simple GRANT and REVOKE statements. 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 [4]. For instance, SQL does not enable conditional limitations on the use of a granted privilege to be specified [4]. Of course, views may also be used to help to control access to information, but combining views and GRANT-REVOKE statements complicates the expression of an access policy.

In the proposed SQL: 1999 standard, language features for expressing RBAC policies are included. However, the proposals are again limited. For instance, SQL: 1999 does not include features for representing temporal authorization policies. Although implementors of SQL database systems may offer more sophisticated options than those included in the SQL:1999 standard, current systems only provide limited options beyond the standard [23].

To augment the expressive power of access control languages, it is possible for a SA to use application programs written in some imperative language (e.g., C). However, the implementation of an access policy in a low-level, procedural language complicates the maintenance of an access policy and makes it difficult for SAs to reason about the consequences and effects of a policy specification.

The need for multipolicy specification using high-level specification languages with well-defined formal semantics has recently been recognised and a number of candidate proposals have been described in the literature [1, 7, 6, 15]. However, these proposals are theoretical in nature and performance measures for implementations of these models are usually missing (albeit [6] is an exception). The contribution of this paper is to describe some *practical* implementations of logic-based specifications of access requirements to protect SQL databases that are accessed over the internet. Another of our contributions is to show how practical implementations of our approach preserves the well-defined semantics upon which the specification of access control policies are based. Evidence for the practicability of our approach is provided in the form of a number of performance measures that we discuss in Section 5. A Java program is used to implement a system that integrates our access control subsystem with client applications and an SQL database. We choose to use Java because of its practical importance.

Recent work on protecting SQL databases has included [4], and [5]. In each of these proposals, SQL is used to implement access control policies for protecting the information in SQL databases, but without using any language-specific constructs for access policy formulation. The decision to use SQL to seamlessly protect SQL databases is a natural one and satisfies the attraction of requiring that a single language be used for implementing access control policies for protecting databases. However, SQL is a relatively low-level language (relative to logic languages), and SQL is not well-suited for proving properties of a policy specification [20]. As such, choosing SQL to formulate access policies for protecting SQL databases is not ideal. Moreover, in [4], the availability of an RDBMS that supports sophisticated *request modification* facilities [11] is assumed. Unfortunately, not all RDBMSs support the features that are required

for the suggested approach, and the formal semantics, upon which the proposal is based, mixes operational and declarative features. In [5], the use of PL/SQL for implementing an access policy for protecting Oracle databases is described. The approach applies only to Oracle SQL databases. Moreover, the proposal involves transforming a formal policy specification from [2] into an equivalent PL/SQL form. However, no formal translation procedure is described. What is more, the implemented code is low-level and hence suffers from the same shortcomings exhibited if applications programs are used to implement access control policies (e.g., reasoning about the formal properties of a policy implementation is complicated).

The rest of the paper is organized thus. In Section 2, some basic notions in access control, RBAC, TRBAC and logic programming are described. In Section 3, the representation of RBAC and temporal RBAC (TRBAC) policies, by using *stratified logic programs* [22], is discussed. These policies are based on the $RBAC_{H2A}$ model that is informally defined in [26] and formally defined in [3]. Henceforth, we refer to the logic programs that implement $RBAC_{H2A}$ ($TRBAC_{H2A}$) policies as *RBAC* (*TRBAC*) programs. In Section 4, we describe our implementation of *RBAC/TRBAC* programs for protecting SQL databases from unauthorized access requests. In Section 5, we give some performance results for our approach. Finally, in Section 6, some conclusions are drawn and suggestions for further work are made.

2. Basic Concepts

The *RBAC* and *TRBAC* programs that we consider in the sequel are represented by using a finite set of *normal clauses* [21]. A normal clause takes the form:

$$H \leftarrow L_1, L_2, \ldots, L_m \qquad (m \geq 0).$$

The *head* of the clause, *H,* is an *atom* and $L_1, L_2, \ldots, L_m$ is a conjunction of *literals* that constitutes the *body* of the clause. The conjunction of literals $L_1, L_2, \ldots, L_m$ must be true (proven) in order for *H* to be true (proven). A literal is an atomic formula (a *positive literal*) or its negation (a *negative literal*); negation in this context is *negation as failure* [10], and the negation of the atom *A* is denoted by *not A*. A clause with an empty body is an *assertion* or a *fact.* In the sequel, we will be principally concerned with stratified programs.

An *RBAC* program *S* is defined on a domain of discourse that includes:

1 A set $\mathcal{U}$ of *users.*

2 A set $\mathcal{O}$ of *objects.*

3 A set $\mathcal{A}$ of *access privileges.*

4 A set $\mathcal{R}$ of *roles.*

The access privileges of interest to us in this paper are defined by the set:

$$\mathcal{A} = \{select, insert, delete, update\}.$$

The semantics of the access privileges in $\mathcal{A}$ are defined thus (where D is an arbitrary SQL database, ΔD is an updated state of D, and θ is a tuple):

- If a user u $(u \in \mathcal{U})$ has the privilege *select* on a set Θ of tuples $(\Theta \subseteq D)$ and $D \models \theta$ then u is permitted to know that $\theta \in \Theta$ is true in D.
- If a user u $(u \in \mathcal{U})$ has the privilege *insert* on a set Θ of tuples $(\Theta \subseteq D)$ then u is permitted to insert $\theta \in \Theta$ into D to generate $\Delta D = D \cup \theta$.
- If a user u $(u \in \mathcal{U})$ has the privilege *delete* on a set Θ of tuples $(\Theta \subseteq D)$ then u is permitted to delete $\theta \in \Theta$ from D to generate $\Delta D = D - \theta$.
- If a user u $(u \in \mathcal{U})$ has the privilege *update* on a set Θ of tuples $(\Theta \subseteq D)$ then u is permitted to change Θ to $\Delta\Theta$ and to generate $\Delta D = (D - \Theta) \cup \Delta\Theta$.

The $\mathcal{U}$, $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{R}$ sets, comprise the (disjoint and finite) sets of user, object, access privilege and role identifiers that form part of the universe of discourse for an *RBAC* program. In this framework we have the following definitions.

Definition 1 *An authorization is a triple* (u, a, o) *that denotes that a user* u $(u \in U)$ *has the a access privilege* $(a \in A)$ *on the object* o $(o \in O)$.

Definition 2 *If a is an access privilege and* o *is an object then a permission is a pair* (a, o) *that denotes that the a access privilege may be exercised on* o.

Definition 3 *A permission-role assignment is a triple* (a, o, r) *that denotes that the permission* (a, o) *is assigned to the role* r.

Definition 4 *A user-role assignment is a pair* (u, r) *that denotes that the user* u *is assigned to the role* r.

For *TRBAC* programs, in addition to elements from the sets $\mathcal{U}$, $\mathcal{O}$, $\mathcal{A}$ and $\mathcal{R}$, we require a set $\mathcal{T}$ of *time points*. We view time as a linearly ordered, discrete set of time points that are isomorphic to the natural numbers.

Definition 5 *A temporal authorization is a 4-tuple* (u, a, o, t) *that denotes that user* u *has the* a *access privilege on object* o *at time* t.

In this paper, we assume that a *(T)RBAC* program defining a *closed policy* [8] is used to protect an SQL database. The implementation of various open policies or any number of hybrid (i.e., open/closed) policies for protecting *D* necessitates that only minor modifications be made to the approach that we describe (see [6]).

In the sequel, the constants that appear in clauses will be denoted by symbols that appear in the lower case; variables will be denoted by using upper case symbols. Moreover, we will use the constants u, o, a, r and t to denote a distinct, arbitrary user, object, access privilege, role and time, respectively. In clauses, we use the variables *U, O, A, R* and *T* to respectively denote a set of users, objects, access privileges, roles and times.

The result that follows is an immediate consequence of a *(T)RBAC* program being a set of stratified clauses.

Proposition 1 *An (T)RBAC program S has a unique 2-valued well-founded model that coincides with the perfect model of S [9].*

Corollary 1 *(T)RBAC programs define a consistent and unambiguous set of authorizations.*

In our representation of *(T)RBAC* programs, functions are only used to express structured terms and do not result in unbounded terms being generated during computation. Moreover, if a *(T)RBAC* program is expressed by using built-in comparison or mathematical operators then we assume that a SA will express these definitions in a *safe* form [27] such that the arguments of a condition involving these operators are bound to constants prior to the evaluation of the condition.

3. Representing *(T)RBAC* Programs

The *RBAC* programs that we describe in this section are based on the specification of *RBAC* as a normal clause program from [1]. In [1], a

user *U* is specified as being assigned to a role *R* by a SA defining a 2-place *ura(U, R)* predicate in an *RBAC* program. For example, $ura(bob, r1) \leftarrow$ is used to record the assignment of the user Bob to the role $r1$. To record that the *A* access privilege on an object *O* is assigned to a role *R,* clause form definitions of a 3-place *pra(A, O, R)* predicate are used. For example, *pra(insert,* $\rho(a, Y), r1) \leftarrow$ expresses that the role $r1$ is assigned the *insert* privilege on the binary relation ρ provided that the pairs *(X, Y)* inserted into ρ are such that $X = a$.

An *RBAC* role hierarchy is represented by a partial order $(R, \geq)$ that defines a seniority ordering $\geq$ on a set of roles *R.* Role hierarchies are used to represent the idea that, unless constraints are imposed, "senior" roles inherit the permissions assigned to "junior" roles in an *RBAC* role hierarchy. Hence, if $r_i \in R$, $r_j \in R$, and $r_i \geq r_j$ then r_i inherits the permissions assigned to r_j.

Following [1], an *RBAC* role hierarchy is expressed in an *RBAC* program by a set of clauses that define a 2-place *senior _to* predicate as the reflexive-transitive closure of an irreflexive-intransitive 2-place ds predicate that defines the set of pairs of roles (r_i, r_j) such that r_i is directly senior to role r_j in an *RBAC* role hierarchy (i.e., r_i is senior to r_j and there is no role $r_k \in \mathcal{R}$ such that r_i is senior to r_k and r_k is senior to r_i).

In clause form logic, *senior_to* is defined in terms of ds thus (where '_' is an anonymous variable):

$$senior\_to(R1, R1) \leftarrow ds(R1, \_).$$
$$senior\_to(R1, R1) \leftarrow ds(\_, R1).$$
$$senior\_to(R1, R2) \leftarrow ds(R1, R2).$$
$$senior\_to(R1, R2) \leftarrow ds(R1, R3), senior\_to(R3, R2).$$

To represent that a user u is active in a role r at a time t, an *active* (u, r) fact is appended to a *(T)RBAC* program whenever u requests to be active in r and this request is allowed. The set of *active* facts in a *(T)RBAC* program at an instance of time t is the set of roles that users have active at time t.

The set of authorization triples defined by an *RBAC* program are expressed by the following clause:

$$permitted(U, A, O) \leftarrow ura(U, R1), active(U, R1), \\ senior\_to(R1, R2), pra(A, O, R2).$$

The *permitted(U, A, O)* clause expresses that a user *U* has the *A* access privilege on an object *O* if *U* is assigned to, and is active in, a role *R*1 that is senior to the role *R*2 in an *RBAC* role hierarchy associated

with the *RBAC* program, and *R*2 is assigned the *A* access privilege on *O*.

Only minor modifications are required to extend *RBAC* programs to *TRBAC* programs. For a *TRBAC* program *S,* the set of authorization triples defined by *S* may be expressed by the following clause:

$$permitted(U, A, O, T) \leftarrow time(T), ura(U, R1, T), active(U, R1), senior\_to(R1, R2), pra(A, O, R2, T).$$

The *permitted(U,A,O,T)* clause expresses that a user *U* has the *A* access privilege on an object *O* at time *T* (extracted from the system clock using *time*(*T*)) if *U* is assigned to a role *R*1 at T, *U* is active in *R*1, *R*1 is senior to the role *R*2 in an *RBAC* role hierarchy defined by the *TRBAC* program, and *R*2 is assigned the *A* access privilege on *O* at *T*.

4. The Practical Implementation of *(T)RBAC* Programs

In this section, we describe the practical implementation of *RBAC* and *TRBAC* programs for protecting SQL databases.

We have adopted a modular approach to developing the software that implements our proposal. There are three principal components in our implementation:

- The Main Program.
- The Security Module.
- The Database System.

The three components above are used with a client application. We refer to an implementation that is based on these components as a *composite system.* It follows that a composite system Σ is defined in terms of a 4-tuple $\Sigma = (\mathcal{M}, \mathcal{S}, \mathcal{D}, \mathcal{C})$ where: $\mathcal{M}$ denotes the main program; $\mathcal{S}$ denotes the security module; $\mathcal{D}$ denotes the database system; and $\mathcal{C}$ denotes a client application. The key features and issues relating to a composite system Σ are briefly outlined below.

The Main Program. The main program $\mathcal{M}$ is written in Java: Java is a suitable language for this type of application because of its comprehensive server programming support (using *Java Servlets* [18]) and its ability to communicate easily with a DBMS using JDBC [19]. In addition, it is easy to access applications written in a variety of other languages (through the *Java Native Interface (JNI)* [17] mechanism).

Java's support for distributed processing means that it will also be well-suited for future developments that we are considering for access control in a distributed DBMS environment.

The $\mathcal{M}$ module acts as a server program that receives access requests from a client application $\mathcal{C}$. The client application is written in HTML for display in a browser window. All user data will be entered via the browser, and all data will be returned to a browser; this is typically the case with e-commerce applications. The principal function of the $\mathcal{M}$ module is to call the $\mathcal{S}$ and $\mathcal{D}$ modules to respectively authorize an access request and to process the request with respect to an SQL database. In the case of a SELECT request involving a query Q on $\mathcal{D}$, the output produced by the $\mathcal{M}$ module is the set Θ of tuples that satisfies Q from $\mathcal{D} \cup \mathcal{S}$; Θ is the answer that is returned to $\mathcal{C}$ by $\mathcal{M}$. Alternatively, if the request for authorization is denied by $\mathcal{S}$, an error message indicating the denial of service is returned to $\mathcal{C}$ by $\mathcal{M}$ and no further action is taken; no request is passed to $\mathcal{D}$ and no DBMS activity occurs.

We have developed two alternative implementations of a session management system. We call these methods $M1$ and $M2$ where:

- $M1$ involves authorizing every database transaction individually, so the duration of a user session is precisely one transaction. This does involve a certain processing overhead, but gives a high level of security where the *RBAC* policy includes either temporal or *dynamic separation of duties* [26] constraints.

- $M2$ establishes a longer session that remains current until either the user terminates the session, or the session is terminated by the system (by, say, a timeout mechanism). The role allocation the user is given is recorded by an entry in a database table used purely for this purpose. The $M2$ method has the additional benefit of being easy to extend to include temporal constraints (i.e., constraints on access that are more sophisticated than a mere inactivity or length-of-session based timeout mechanism) and a dynamic separation of roles constraint. The latter option is straightforward to implement as our $\mathcal{S}$ module can directly consult the database table in which current role allocations are stored.

The Security Module. Within the $\mathcal{S}$ module, we use XSB [24] to implement the logic program that defines the access control applicable to an SQL database to be protected. XSB offers excellent performance that has been demonstrated to be far superior to that of traditional Prolog-based systems [25]. Calls to XSB from $\mathcal{M}$ are handled by the YAJXB [29] package. YAJXB makes use of Java's JNI mechanism to

invoke methods in the C interface library package supplied by XSB. It also handles all of the data type conversions that are needed when passing data between C and Prolog-based applications. YAJXB effectively provides all the functionality of the C package within a Java environment.

Although we have used YAJXB in our implementation of composite systems, we note that a number of alternative options exist. Amongst the options that we considered for implementing composite systems were:

- Interprolog [14].
- One of the Java-based Prolog interpreters currently available (e.g., JavaLog [16]).
- a Sockets-based, direct communication approach [12].

Each of the above approaches has its own distinct drawbacks when compared with the approach that we adopted. Interprolog does work with XSB, so we could still take advantage of the latter's performance capabilities. However, Interprolog is primarily a Windows-based application. All of our development was done on a Sun Sparc/Solaris system; YAJXB, though primarily configured for Linux, compiles easily on Solaris. JavaLog was discounted because we felt that it did not offer sufficient performance for the types of implementations and volumes of data involved in practical applications of our approach. Finally, using sockets would give us a less flexible application because it would involve considerably more application-specific coding. Overall, we felt that the straightforwardness of the YAJXB interface makes it preferable to the Interprolog approach so far as interfacing with XSB is concerned. Moreover, XSB's highly developed status and excellent performance make it more desirable in this context than a Java/Prolog hybrid.

Once XSB has been successfully invoked by $\mathcal{S}$, XSB loads a program that contains the Prolog expression of an *RBAC* program (see above). This is used to determine whether the access requests made via the client application $\mathcal{C}$ are permitted (by $\mathcal{S}$) or not.

The C library allows the full functionality of XSB to be used. A variety of methods for passing Prolog-style goal clauses to XSB exists. However, we generally found that YAJXB's string method worked well. This method involves constructing a string σ in a Java String type variable, and using the *xsb_command_string* function (or similar) to pass σ to XSB. This approach allows any string that could be entered as a command when using XSB interactively to be passed to XSB by $\mathcal{S}$. YAJXB creates an interface object. The precise method of doing this is a

call like

$$i = core.xsb\_command\_string(command.toString());$$

where the assignment, as one would expect, handles the returned error code. More sophisticated methods would allow for the return of data too; though this is not necessary where authorization is merely confirmed or denied.

The key technical results for query evaluation on SQL databases protected by using $\mathcal{D} \cup \mathcal{S}$ follow directly from the soundness and completeness results for SLG-resolution applicable to stratified theories. As $\mathcal{S}$ is a function-free stratified program and YAJXB only passes facts to XSB it follows that every authorized access request is provable by SLG-resolution and no unauthorized access is provable by (safe) SLG-resolution. These results extend to *TRBAC* programs.

Database System. Although $\mathcal{D}$, in a composite system, may be any SQL database, we have used Oracle in implementations of our system (because of its widespread use within industry). A composite system could easily be adapted to apply to a large number of existing Oracle applications. For the interface between $\mathcal{M}$ and $\mathcal{D}$, we have used JDBC [19]. JDBC is now a well-established technology and has the additional advantage that JDBC drivers exist for numerous DBMS packages. It follows that, with minimal modifications, a composite system could be used with any DBMS with which JDBC drivers may be used.

5. Performance Measures

In any system where data is accessed over the Internet, by far the biggest time overhead will be caused by communications costs. As our model does not introduce any additional traffic (i.e., the data sent by, and returned to, a remote user is the same as it would be if our authorization model were not used), this component of performance cost remains unchanged, and we do not therefore consider any specific time delay values. Similarly, the overheads associated with accessing the DBMS are unchanged, and we again do not consider any specific values.

The performance cost that we have added to a transaction is the one of calling the subsystem $(\mathcal{S})$ that we use to authorize access requests. Although the costs of using $\mathcal{S}$ are minimal when compared with communication costs, we have conducted various performance tests on an *RBAC* program that we use to protect SQL databases from unauthorized access requests. Our *RBAC* program includes a definition of a 53 role *RBAC* role hierarchy that has been represented by using a set of facts to represent all pairs of roles in the *senior_to* relation (a to-

tal of 312 *senior_to* facts). For our SQL tests, we have used the data from [5]. There is one user, one *ura* rule, 8 database objects (tables) containing a total of 432,261 tuples, and 720 *pra* rules. It is sufficient for test purposes to use one user to demonstrate a worst-case use of the access control information in an *RBAC* program. This worst case test involves assigning a user u to the unique top element in the *RBAC* role hierarchy, such that u has complete access to all of the tables used in the test queries. The permissions are assigned to the unique bottom element in the *RBAC* role hierarchy. Hence, our testing involves the maximum amount of multiple upward inheritance of permissions.

The experiments were performed using XSB Version 2.5 on a Sun Ultra 60 server (2 450MHz CPUs and 1GB RAM) running Solaris. Typically, the time taken to evaluate an authorization request is less than a hundredth of a second. Furthermore, where a user's request for data is not authorized, no database access takes place at all and no processing costs are incurred. A summary of our results for "fixed" costs is given in Table 1. By "fixed" we mean the fixed overheads of invoking XSB.

Operation	*Time (seconds)*
Start XSB	*0.07*
XSB loads and compiles RBAC program	*1.285*
XSB loads precompiled RBAC program	*0.01*

Table 1. Fixed Costs

The figures in Table 1 are averages: each operation was performed five times. On first load of the Prolog program (which in this case represents our *RBAC* policy), XSB compiles the code and stores the compiled version. Subsequently, if the program has not changed since the last compilation, XSB loads the compiled version, with a significant reduction in the load time.

The performance times that we give have been obtained using XSB's *statistics* package. XSB gives times for CPU usage in seconds, accurate to two decimal places.

Table 2 shows the results we obtained for a number of tests of access requests (averaged over ten runs). We performed worst case tests (see above) and, for comparison, best case tests (where there is no upward inheritance of permissions). For each, we tested with data that would give both possible outcomes for the requested database operation.

Case	*Outcome*	*Time (seconds)*
worst	*permitted*	*0.002*
worst	*denied*	*0.001*
best	*permitted*	*0.002*
best	*denied*	*0.0*

Table 2. Variable Costs

Given the magnitude of these figures, it is possible that measuring inaccuracies render the small differences between them difficult to evaluate. We confine ourselves to noting that the query execution time is minimal when compared with the time taken to start XSB and load the program. The total time required to authorize a database access request, if the RBAC program has not been updated since the last time it was compiled, is about a tenth of a second. This compares favourably with the results we obtained in [5], where a PL/SQL-based implementation of our *RBAC* model took 1.08 seconds.

In contrast to user queries on $\mathcal{D}$, it should be noted that a SA may evaluate administrative review queries with respect to an $RBAC_{H2A}$ program $\mathcal{S}$ directly. For example, to generate the set of users assigned to the role $r1$, in the process of performing a *user-role review* [26], a SA simply needs to use SLG-resolution to evaluate the goal clause ? – $ura(U, r1)$ with respect to $\mathcal{S}$.

For completeness, we performed a number of such queries; the results are given in Table 3. We believe that the apparently identical times taken for almost all of the queries shows that the computation time was too small for XSB to accurately record.

Number of ura *facts*	*Number of Users assigned to role* r1	*Time (seconds)*
300	*1*	*0.01*
300	*6*	*0.01*
300	*40*	*0.01*
600	*6*	*0.01*
600	*65*	*0.04*

Table 3. Retrieval times for test database D.

6. Conclusions and Further Work

We have shown how the information in SQL databases may be protected from unauthorized access requests by using *RBAC* and *TRBAC* programs. The high-level formulation of an access policy as a logic program makes it relatively easy for a SA to express an access policy, to reason about its effects and to maintain it. Moreover, it is possible to

use this specification of policy in an implementation of composite systems. We have demonstrated that the access policies that we use for protecting SQL databases may be efficiently implemented.

In future work, we intend to investigate how constraint checking on $\mathcal{S}$ may be incorporated into the approach that we have described here.

References

[1] Barker, S., Data Protection by Logic Programming, *1st International Conference on Computational Logic,* LNAI 1861, 1300-1313, Springer, 2000.

[2] Barker, S., $TRBAC^N$: A Temporal Authorization Model, *Proc. MMMANCS International Workshop on Network Security,* in V. Gorodetski, V. Skormin, and L. Popyak (Eds.), Lecture Notes in Computer Science 2052, Springer, 178-188, 2001.

[3] Barker, S., Protecting Deductive Databases from Unauthorized Retrieval and Update Requests, *Journal of Data and Knowledge Engineering,* Elsevier, 293-315, 2002.

[4] Barker, S., and Rosenthal, A., Flexible Security Policies in SQL, DBSec 2001, 187-199, 2001.

[5] Barker, S., Douglas, P. and Fanning, T., *Implementing RBAC Policies in PL/SQL,* DBSec 2002.

[6] Barker, S., and Stuckey, P., Flexible Access Control Policy Specification with Constraint Logic Programming, ACM Trans. on Information and System Security, 6, 4, 501-548, 2003.

[7] Bertino, E., Catania, B., Ferrari, E., and Perlasca, P., A System to Specify and Manage Multipolicy Access Control Models, *Proc. IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002),* 116-127, 2002.

[8] Castano, S., Fugini, M., Martella, G., and Samarati, P., *Database Security,* Addison-Wesley, 1995.

[9] Chen, W., and Warren, D., A Goal-Oriented Approach to Computing the Well-Founded Semantics, *J. Logic Programming,* 17, 279-300, 1993.

[10] Clark, K., Negation as Failure, in H.Gallaire and J. Minker(Eds), *Logic and Databases,* Plenum, NY, 293-322, 1978.

[11] Date, C., *An Introduction to Database Systems (7th Edition),* Addison-Wesley, 2000.

[12] Donahoo, M. and Calvert, K, *The Pocket Guide to TCP/IP Sockets,* Morgan Kaufmann, 2001.

[13] Ferraiolo, D., Gilbert, D., and Lynch, N., An Examination of Federal and Commercial Access Control Policy Needs, *Proc. 16th NIST-NSA National Computer Security Conference,* 107-116, 1993.

[14] InterProlog by Declarativa. www.declarativa.com/InterProlog/default.htm

[15] Jajodia, S., Samarati, P., Sapino, M., and Subrahmaninan, V., Flexible Support for Multiple Access Control Policies, *ACM* TODS,26, 2, 214-260, 2001.

[16] The ISISTAN Brainstorm Project: JavaLog. www.exa.unicen.edu.ar/~azunino/javalog.html

[17] Java Native Interface, *Sun Microsystems.* java.sun.com/products/

[18] Java Servlet Technology: Implementations and Specifications, *Sun Microsystems.* java.sun.com/products/jdk/1.2/docs/guide/jni

[19] The JDBC API, *Sun Microsystems.* java.sun.com/products/jdbc

[20] Libkin, L, The Expressive Power of SQL, *Proc. ICDT,* 1-21, 2001.

[21] LLoyd, J., *Foundations of Logic Programming,* Springer, 1987.

[22] Przymusinski, T., Perfect Model Semantics, *Proc. 5th ICLP,* MIT Press, 1081-1096, 1988.

[23] Ramaswarmy, C., and Sandhu, R., Role-Based Access Control Features in Commercial Database Management Systems, *Proc. 21st National Information Systems Security Conference,* 503-511, 1998.

[24] Sagonas, K., Swift, T., Warren, D., Freire, J., Rao, P., The XSB System, Version 2.0, Programmer's Manual, 1999.

[25] Sagonas, K., Swift, T. and Warren, D., XSB as an Efficient Deductive Database Engine , *ACM SIGMOD Proceedings,* p512, 1994.

[26] Sandhu, R., Ferraiolo, D., and Kuhn, R., The NIST Model for Role-Based Access Control: Towards a Unified Standard, *Proc. 4th ACM Workshop on Role-Based Access Control,* 47-61, 2000.

[27] Ullman, J., *Principles of Database and Knowledge-Base Systems: Volume 1,* Computer Science Press, 1990.

[28] Van Gelder, A., Ross, K., and Schlipf, J., The Well-Founded Semantics for General Logic Programs, *J. ACM,* 38(3), 620-650, 1991.

[29] Decker, S., YAJXB, www-db.stanford.edu/~stefan/rdf/yajxb