An Authorization Model and Its Formal Semantics

Elisa Bertino,[1] Francesco Buccafurri,[2] Elena Ferrari,[1] Pasquale Rullo[3]

[1] Dipartimento di Scienze dell'Informazione
Università di Milano - Milano, Italy
{bertino,ferrarie}@dsi.unimi.it

[2] Dipartimento DIMET
Università di Reggio Calabria, Reggio Calabria, Italy
bucca@ns.ing.unirc.it

[3] Dipartimento di Matematica
Università della Calabria, Arcavacata di Rende, Rende, Italy
rullo@ns.ing.unicr.it

Abstract. This paper describes an advanced authorization mechanism based on a logic formalism. The model supports both positive and negative authorizations. It also supports derivation rules by which an authorization can be granted on the basis of the presence or absence of other authorizations. Subjects, objects and authorization types are organized into hierarchies, supporting a more adequate representation of their semantics. From the authorizations explicitly specified, additional authorizations are automatically derived by the system based on those hierarchies. The combination of all the above features results in a powerful yet flexible access control mechanism.
The specification language of the system is an extension of Ordered Logic with ordered domains. This is an elegant yet powerful formalism whereby the basic concepts of the authorization model can be naturally formalized. Its semantics is based on the notion of stable model and assigns, to a given set of authorization rules, a multiplicity of (stable) models, each representing a possible way of assigning access authorizations. This form of non-determinism entails an innovative approach to enforce access control: when an access request is issued, the appropriate model is chosen on the basis of the accesses currently under execution in the system.

1 Introduction

The introduction of an access control system within any organization entails two main tasks. The first is the identification and specification of suitable *access control policies.* An access control policy establishes for each subject the actions he/she can perform on which object within the system under which circumstances. The second task is the development of a suitable *access control mechanism* implementing the stated policies. Many advanced applications, such as workflow systems, and computer-supported cooperative work, have articulated and rich access control requirements. These requirements cannot be adequately supported by current access control mechanisms which are tailored to few, specific policies. In most cases, either the organization is forced to adopt the specific policy built-in into the access control mechanism at hand, or access control policies must be implemented as application programs. Both situations are clearly unacceptable. A possible solution to this problem is the development of advanced access control mechanisms, which extend the expressive power of access control mechanisms currently available in commercial DBMSs and research prototypes. Extending the expressive power of existing authorization models has, as its counterpart, an increase of the complexity of the model. A method to manage such increased complexity is to provide a formal semantic foundation for the model. In this paper we present a step in this direction by proposing an authorization mechanism based on a logic formalism. The model is based on the closed policy with negative authorizations and the enforcement of the "strongest authorization takes precedence" principle. Our model also provides the possibility of specifying *derivation rules* by which new authorizations can be derived on the basis of the presence or absence of other authorizations. Authorizations can be granted to a single user as well as to a group or to a role. Subjects, objects, and authorization types are organized into hierarchies, supporting a more adequate representation of their semantics. Authorizations automatically propagate along these hierarchies according to a set of *propagation rules*. The combination of all the above features results in a powerful yet flexible authorization model.

The authorization specification language of the system is an extension of Ordered Logic (OL) [5, 6, 13] with ordered domains. This is an elegant yet powerful formalism whereby the basic concepts of the authorization model can be directly represented. Its semantics is an extension of the stable model semantics of logic programs [10, 20] to deal with hierarchies, ordered domains, and true negation. According to this semantics, more stable models can be assigned to a given set of authorization rules, each representing a consistent set of assignments of access authorizations. This form of non-determinism entails an innovative approach to enforce access control: when an access request is issued, the set of authorization rules against which the access request must be verified is chosen on the basis of the accesses currently under execution in the system.

The development of flexible authorization models has been addressed in other papers [2–4, 9, 11, 19]. Most of the previous proposals, however, have one or more of the following shortcomings: (i) only group [4, 9, 11] or role hierarchies [19] are supported; (ii) only propagation rules [9, 19] or derivation rules [2, 3, 11] are supported; (iii) very limited form of derivation rules are supported [2, 3, 9] (iv) very restrictive conflict resolution policies with no exceptions are adopted [2–4]; (vi) object hierarchies are not supported [2–4, 11]; (vii) access mode hierarchies are not supported [2–4, 9, 11]. By contrast, our model integrates most of the features of the abovementioned models into a common framework and provides a semantic foundation for them. Additionally, our model has an articulated conflict resolution policy, to deal with authorization conflicts and exceptions, and an innovative approach to enforce access control.

From the side of logic formalisms for security specifications, Jajodia et al. [12] propose a logic language for expressing authorization rules and show how this language can express most of the access control policies proposed so far. Programs that can be written in this language are a subset of stratified Datalog programs and therefore they are able to express only a limited set of authorization specifications. By contrast, in this paper we propose a very general language to express authorization specifications without syntactic restrictions (like stratification). Hence, we do not restrict ourselves to the consideration of programs having a unique model rather we allow a multiplicity of models to be associated with a given program. To deal with this multiplicity, we have developed a strategy that, each time an access request is submitted to the system, allow us to select the appropriate model against which the access request must be checked. A logic language, based on modal logic, has been proposed by Abadi et al. in [1]. However, their logic is mainly devoted to express concepts such as roles, delegation of authorizations, or the operation of certain protocols. A general logic language for expressing authorization rules has also been proposed by Woo and Lam in [22]. Although their language is very expressive, it suffers from several drawbacks. The most important is that it is not always possible to decide whether an access request can be authorized or not, because of conflicting authorizations, and no mechanisms are provided to deal with such inconsistencies.

The remainder of this paper is organized as follows. Section 2 presents the basic elements of our authorization model. Section 3 provides the logic framework to express the various components of our model. Section 4 deals with access control. Finally, Section 5 concludes the paper and outlines future work.

2 Overview of the Authorization Model

In this section we illustrate our authorization model.

2.1 Subjects, Objects and Privileges

Our model relies on three basic components.

The first component is a set of subjects S to which authorizations are granted. Subjects can be *single users* (i.e., elements of a set U), *groups* (i.e., elements of a set G) or *roles* (i.e., elements of a set R). Roles are named collection of privileges and represent organizational agents intended to perform certain job functions within an organization. Users in turn are assigned appropriate roles based on their qualification. To enable users to successfully execute their tasks, each role has some authorizations associated with it. A user can be authorized to *play* several roles. When a user takes on a role, he/she is allowed to exercise all the privileges associated with the role. Moreover, a role may be played by several users.[1] Usually, roles within an organization are hierarchically organized. For

[1] We assume that there is some mechanism in place that associates users with roles.

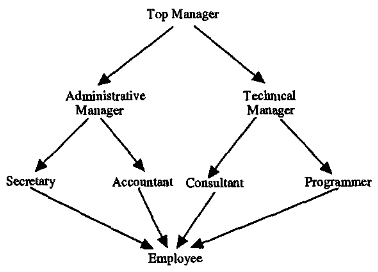

Fig. 1. An example of role hierarchy

instance the role Manager is considered at higher level than the role Accountant. To reflect this situation we assume that roles are organized according to a partial order, denoted as $\prec_R$. Therefore, the set R together with the $\prec_R$ relation forms a hierarchy,[2] referred to as *role hierarchy*. Such hierarchy reflects the organizational position of roles within a given organization. Let $r_i, r_j \in R$ be roles. We say that r_i *dominates* r_j in the hierarchy ($r_i \prec_R r_j$), if r_i precedes r_j in the ordering. An example of role hierarchy is illustrated in Figure 1.

Groups are sets of users and/or roles. A user or role may belong to several groups. A partial order is defined on the set G of groups, denoted as $\prec_G$, which represents the group-subgroup relationship. Given two groups g_i and g_j, $g_i \prec_G g_j$ if and only if g_j is a subgroup of g_i. The set G and the $\prec_G$ relation form a hierarchy, referred to as *group hierarchy*.

The second component of our authorization model is a set of objects O, denoting the resources to be protected. On the set of objects O a partial order is defined, reflecting the way objects are organized in terms of other objects. Given two objects o_i and o_j, we say that o_j is a *component* of object o_i, denoted $o_i \prec_O o_j$, if o_i precedes o_j in the ordering. The set O and the $\prec_O$ relation form a hierarchy, referred to as *object hierarchy*. An example of object hierarchy is illustrated in Figure 2.

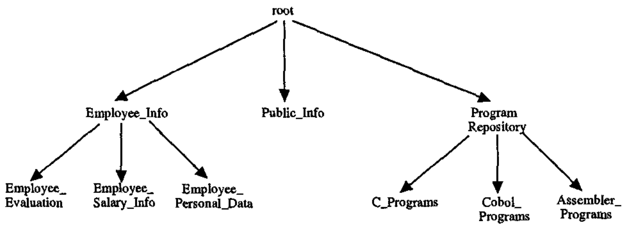

Fig. 2. An example of object hierarchy

[2] By hierarchy we mean a poset $(S, \prec)$, where S is a set and $\prec$ is a partial order over S.

The last component of our model is a set of privileges P, denoting the access modes subjects can exercise on the objects in the system. In real situations, interactions exist among privileges. For instance, it is reasonable to assume that the `write` privilege is *stronger* than the `read` privilege. For this reason, also the set of privileges P is organized into a hierarchy, called *privilege hierarchy*, by means of a partial order $\prec_P$. We say that a privilege p_j is a *sub-privilege* of privilege p_i if and only if the relation $p_i \prec_P p_j$ holds. This means that privilege p_j is subsumed by privilege p_i.

2.2 Positive and Negative Authorizations

Authorizations can be either positive or negative. A positive authorization states that a subject is authorized for a given privilege on a given object, whereas a negative authorization states that a subject must be denied access to a given object. In the following we use the notation (s, o, p, g) to denote positive authorizations, and the notation $\neg(s, o, p, g)$ to represent negative authorizations, where $s \in S$, $o \in O$, $p \in P$ and $g \in U \cup R$ (a description of the formal model is reported in Section 3). Tuple (s, o, p, g) states that subject g authorizes subject s to exercise privilege p on object o, whereas $\neg(s, o, p, g)$ states that g prevents s from exercising privilege p on o.

Note that roles can appear as grantor in an authorization. The reason is that, in our model, a user can connect to the system either as a role or as a single user (we refer to the role with which the user connects as the user's *active role*).[3] When a user logs into the system as a role, all the actions he/she performs (such as granting or revoking authorizations) are considered as if they are performed directly by the user's active role rather than by the user itself.

Example 1. Authorization `(Employee,read,Public_Info,Bob)`, granted by `Bob`, authorizes the role `Employee` to read the object `Public_Info`, whereas authorization $\neg$`(Employee,read,Employee_Evaluation,Manager)`, granted by the role `Manager`, prevents the role `Employee` from reading `Employee_Evaluation`.

2.3 Derivation Rules

Beside explicit authorizations, the model supports the specification of *derivation rules* stating the permission or denial for subjects to exercise privileges on objects on the basis of the presence or absence of other permissions or denials. Derivation rules can be used to derive both positive and negative authorizations, and represent a compact way to specify a set of authorizations.

Example 2. The following are examples of derivation rules that can be expressed in our model:

1. User `Ann` is denied to exercise the `execute` privilege on `C_Programs` if the role `Programmer` has an authorization to `write` it.

[3] For simplicity, we make the assumption that a user is allowed to take on only one role in a session

2. Users `Ann`, `Bob`, and `John` must have exactly the same authorizations on all the objects.
3. The roles `Administrative Manager` and `Technical Manager` must have mutually exclusive write authorizations on object `Employee_Evaluation`.

The language to express derivation rules will be illustrated in Section 3. As it will be clear in that section, the presence of negation in derivation rules (whereby new authorizations are derived based on the absence of others) may cause the existence of several sets of access authorizations. Consider, for instance, the third rule in the above example, where the authorization to write object `Employee_Evaluation` can be derived for the role `Administrative Manager` provided that the role `Technical Manager` is not authorized to write the same object, and vice versa. Clearly, two sets of assignments can be derived in this case, one in which only `Administrative Manager` has the permission to write `Employee_Evaluation` and the other in which only `Technical Manager` has this permission.

2.4 Derivation of Authorizations along Role, Group, Object, and Privilege Hierarchies

Our model supports two different types of implicit authorizations: the first, discussed in the previous section, consists of authorizations derived from the derivation rules specified by the user. The second derives from the role, group, object and privilege hierarchies. Authorizations automatically propagate along these hierarchies according to a set of *propagation rules.* These propagation rules apply to both explicit authorizations and authorizations implicitly given through derivation rules.

As far as the role hierarchy is concerned, we consider the following propagation rules: *1)* a positive authorization given to a role r propagates to all roles which precede r in the role hierarchy (that is, to all roles r' such that $r' \prec_R r$); *2)* a negative authorization given to a role r propagates to all the roles following r in the role hierarchy (that is, to all roles r' such that $r \prec_R r'$).

Note that the above propagation rules are directly implied by the semantics given to the $\prec_R$ relationship and they are used in practice by commercial DBMSs (such as [18]) supporting the notion of role. Since the lower is the level of a role in the role hierarchy, the lower is its position within an organization, it is reasonable to assume that the access privileges given to a role subsume the access privileges given to all roles with a lower position in the hierarchy.

Example 3. Consider the role hierarchy in Figure 1. From the explicit authorizations:

`(Administrative Manager,write,Employee_Personal_Data,Bob)`,
$\neg$`(Administrative Manager,execute,Program_Repository,John)`
the following authorizations are derived:
`(Top Manager,write,Employee_Personal_Data,Bob)`,
$\neg$`(Secretary,execute,Program_Repository,John)`,
$\neg$`(Accountant,execute,Program_Repository,John)`,
$\neg$`(Employee,execute,Program_Repository,John)`.

Moreover, we assume that an authorization, either positive or negative, given to a group propagates to all the members of the group. Thus, if a user logs into the system as a role he/she has all the authorizations explicitly given to the role (or derived through propagation and/or derivation rules) and all the authorizations of the groups to which the role belongs to; if he/she connects as a single user, then he/she has all his/her personal authorizations and all the authorizations of the groups to which he/she belongs to.

Similarly to the role hierarchy, authorizations propagate along the privilege hierarchy according to the following rules: *1)* a positive authorization for privilege p on object o implies a positive authorization on o for all the privileges following p in the privilege hierarchy; *2)* a negative authorization for p on o implies a negative authorization on o for all the privileges preceding p in the privilege hierarchy.

Example 4. Consider the authorizations of Example 3 and suppose that `write` $\prec_P$ `read`. The following additional authorizations are derived:

`(Administrative Manager,read,Employee_Personal_Data,Bob)`,
`(Top Manager,read,Employee_Personal_Data,Bob)`.

Finally, authorizations granted on a given object propagates to all the objects which are direct or indirect components of it.

Example 5. Consider the object hierarchy illustrated in Figure 2. From the authorizations of Examples 3 and 4 we derive a negative authorization for `Administrative Manager`, `Secretary`, `Accountant`, and `Employee` to execute `C_Programs`, `Cobol_Programs`, and `Assembler_Programs`.

2.5 Conflict Resolution Policy

In our model conflicts may arise due to the simultaneous presence of a positive and a negative authorization with the same subject, object and privilege. We do not consider the simultaneous presence of conflicting authorizations as an inconsistency, rather we define a conflict resolution policy which is based on the notion of *strongest authorization.* The conflict resolution policy enforced by our model keeps into account:

– the grantors of the conflicting authorizations, that is, their relative positions in the role hierarchy; the authorization granted by the higher level grantor is considered as prevailing. Clearly this conflict resolution mechanism applies when both the conflicting authorizations are granted by a role;
– the object hierarchy, in that when conflicts are not solved by the role hierarchy, the authorization specified at a lower level in the object hierarchy is considered as dominating;
– the sign of the authorizations, since when conflicts cannot be solved by considering the role and/or object hierarchy, we consider as prevailing negative authorizations.

Example 6. Consider the following authorizations:

(Employee,write,Employee_Info,Top Manager),
$\neg$(Employee,write,Employee_Info,Administrative Manager),
$\neg$(Consultant,execute,Program_Repository,Technical Manager),
(Consultant,execute,Program_Repository,Tom).

- (Employee,write,Employee_Info,Top Manager) prevails over $\neg$(Employee,write,Employee_Info,Administrative Manager) since Top Manager $\prec_R$ Administrative Manager;
- $\neg$(Consultant,execute,Program_Repository,Technical Manager) prevails over (Consultant,execute,Program_Repository,Tom). In this case the conflict cannot be solved neither by the role nor by the object hierarchy. Thus, the negative authorization is considered as prevailing.

2.6 Strong Authorizations

The authorization model we have defined so far is characterized by a high degree of flexibility. However, this flexibility implies some loss of control from the owner of the object. As an example, let us suppose that a user, say Bob, when playing the role Employee, writes some information into one of his objects, say o_1, and that he does not want to disclose this information to Alice. He therefore grants a negative authorization for the read privilege to Alice on object o_1. However, the negative authorization issued by Bob does not always ensure that Alice is forbidden to read object o_1. For instance, if a user playing the role Manager grants Alice a positive authorization for the read privilege on object o_1, this latter authorization overrides the authorization granted by Bob.

To overcome this drawback we introduce the concept of *strong authorizations*, that is, authorizations not admitting exceptions with respect to the role hierarchy. Strong authorizations can be specified either explicitly or through derivation rules. The basic idea is that: *i)* strong authorizations cannot be overridden by weak authorizations; *ii)* conflicts among strong authorizations are solved in favor of the authorization specified at the lower level in the object hierarchy. If the objects on which the authorizations are specified are not related by the $\prec_O$ hierarchy, the negative authorization is considered as prevailing.

Note that the notion of strong authorization we propose is different from the one adopted by other models (see, for instance, [19]) supporting weak and strong authorizations. In these models, strong authorizations can never be overridden by other strong authorizations. This implies that the insertion of a strong authorization must be rejected by the system if it conflicts with an existing strong authorization. This clearly prevents strong authorizations to be granted through derivation rules. To avoid these shortcomings, we allow strong authorizations to be overridden by other positive or negative strong authorizations, under specific circumstances. However, this possibility does not invalidate the usefulness of strong authorizations: a user can retain complete control over an object o by issuing a negative strong authorization on all the direct/indirect components of o which do not have any component object.

3 Formal Model

In this section we present the language whereby authorizations are specified in our system and give its formal semantics. The language is based on an extension of Ordered Logic (OL) [5, 6, 13] by ordered domains.

3.1 The Authorization Specification Language

An authorization program encodes the rules whereby authorizations are granted within a given system. The components of such a system, namely, subjects, objects and privileges, along with their relationships, are modeled by the notion of *program domain*, that we next formally define.

Definition 1. (Program Domain) A *(program) domain* consists of the following components:

1. A countable set S of labels, called *subject identifiers.* This set is partitioned into three subsets, namely, G (*groups*), R (*roles*) and U (*users*). On G the partial order relation $\preceq_G$ is defined. The set R contains a built-in role denoted $\top_R$. The set $R \cup U$ is partially ordered by $\preceq_R$ in such a way that $\top_R \preceq_R a$, for all $a \in R \cup U$ and, further, $a \preceq_R b$, with $a \neq \top_R$, implies $a, b \in R$ (the element $\top_R$ is used to model strong authorizations). We denote by $\prec_R$ the reflexive reduction of $\preceq_R$.[4] Further, the following functions are given: *User_Groups* : $G \to 2^U$ that, given a group g, returns the users members of g; and *Role_Groups* : $G \to 2^R$ that, given a group g, returns the roles members of g.
2. A countable set O of labels, called *object identifiers.* O is partially ordered by $\preceq_O$ and the poset $(O, \preceq_O)$ has a top element denoted $\top_O$, that is, an element such that $\top_O \preceq_O o$, for all $o \in O$. This element is used to define properties that must hold on all the elements of O. The partial order $\preceq_O$ models a *part-of* relation among objects. We denote by $\prec_O$ the reflexive reduction of $\preceq_O$.
3. A countable set P of labels, called *privilege identifiers.* P is partially ordered by $\preceq_P$. We denote by $\prec_P$ the reflexive reduction of $\preceq_P$.

From now on, we assume that a domain D has been fixed. Further, we assume that the following sets are given: *1)* A set Π of *predicate symbols* of two types: *built-in* and *user-defined.* There is a unique built-in predicate symbol, namely, *auth*, of arity 3, which has type $S \times P \times (R \cup U)$. User-defined predicate symbols are untyped and have a fixed arity; *2)* a set Λ of *variable symbols.*

We next define our language based on the fixed domain D (from which constants are taken) and the above defined sets Π and Λ (thus, henceforth, every notion is implicitly defined on D, Π and Λ).

A *term* is either a constant (of D) or a variable (in Λ). An *atom* is of the form: $p(t_1, ..., t_n)$, where p is a predicate symbol (in Π), n is the arity of p and $t_1, ..., t_n$ are terms. If p is built-in (i.e., $p = auth$) and its type is $\tau_1 \times \tau_2 \times \tau_3$,

[4] Given a partial order R, the reflexive reduction of R is $\{(a, b) \in R | a \neq b\}$.

then, for each t_i which is a constant, $t_i \in \tau_i$ holds. A *simple literal* is either a *positive literal* Q or a *negative literal* $\neg Q$, where Q is an atom and $\neg$ is the *true negation symbol.* A *referential literal* is of the form $o.L$, where $o \in O$ and L is a simple literal. A *literal* is either a simple or a referential literal. Two simple (resp. referential) literals are *complementary* if they are of the form Q and $\neg Q$ (resp. $o.Q$ and $o.\neg Q$). Given a literal L, we denote its complementary literal by $\neg.L$. Two literals are *conflicting* if they are of the form $auth(s,p,g)$ and $\neg auth(s',p',g')$ (resp. $o.auth(s,p,g)$ and $o.\neg auth(s',p',g')$), where $s = s'$ and $p = p'$.

A *clause* r is an expression of the form:

$$H \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m \qquad n \geq 0, m \geq 0$$

where H, $A_1, ..., A_n$ and $B_1, ..., B_m$ are literals and *not* is the *negation by failure* symbol [15]. H is the *head* of r, whereas $A_1, ..., A_n, not\ B_1, ..., not\ B_m$, $n \geq 0, m \geq 0$, is the *body* of r. Note that the head H may be a negative literal. We often denote the head literal of r by $Head(r)$ and the body of r by $Body(r)$.

Definition 2. (Program Rule) A *(program) rule* is a pair $\langle o, r\rangle$ where $o \in O$ and r is a clause such that $Head(r)$ is a simple literal. A program rule $\langle o, r\rangle$ whose head predicate symbol is *auth* is called *authorization rule.* If the body of r is the empty conjunction, then $\langle o, r\rangle$ is an *explicit* authorization; otherwise, it is a *derivation rule* (whereby *implicit* authorizations are derived).

An authorization rule $\langle o, r\rangle$, where $Head(r)$ is of the form, say, $auth(s,p,g)$ (resp. $\neg auth(s,p,g)$), is used to express a positive (resp. negative) authorization for privilege p granted by g to subject s on object o.

Example 7. The rule: $\langle$`Public_Info`, $\neg auth$(`Employee`, `read`, `Bob`) $\leftarrow\rangle$ encodes an explicit negative authorization to `read` `Public_Info` granted by `Bob` to the role `Employee`. With the rules:
r_1: $\langle$`C_Programs`, $auth$(`Amy`, `read`, `Tom`) $\leftarrow$ *not others_subj_has_write_auth*$\rangle$
r_2 : $\langle$*others_subj_has_write_auth* $\leftarrow$ `C_Programs`.$auth(X,$ `write`$, Y), X \neq$ `Amy`$\rangle$
`Tom` authorizes `Amy` to `read` object `C_Programs` provided that nobody else has an authorization to `write` it.

Definition 3. (Authorization Program) An *(authorization) program* $\mathcal{P}$ is a finite set of program rules. We call each maximal subset of rules in $\mathcal{P}$ having the same object identifier a *component* of $\mathcal{P}$.

For simplicity, and without loss of generality, we assume that user-defined predicates are local to components.

Example 8. Consider the program $\mathcal{P}$ consisting of the following rules:

r_1: $\langle$ `Employee_Evaluation`, *auth*`(Administrative Manager,write,Top Manager)` $\leftarrow$ *not auth*`(Technical Manager,write,Top Manager)`$\rangle$
r_2: $\langle$ `Employee_Evaluation`, *auth*`(Technical Manager,write,Top Manager)` $\leftarrow$ *not auth*`(Administrative Manager,write,Top Manager)`$\rangle$

r_3: $\langle$ `Program_Repository`, *auth*(`Bob,read,Ann`) $\leftarrow$ $\rangle$
r_4: $\langle$ `Employee_Salary_Info`, *auth*(`Accountant,write,Top Manager`)
$\leftarrow$ *not* `Employee_Evaluation`.*auth*(`Administrative Manager,write,`X)$\rangle$

Rules r_1 and r_2 express that the role `Top Manager` prevents roles `Administrative Manager` and `Technical Manager` to simultaneously have the `write` privilege on `Employee_Evaluation`.

3.2 Stable Model Semantics

Throughout this section, we assume that an authorization program $\mathcal{P}$ is given.

The Base $B_{\mathcal{P}}$ of $\mathcal{P}$ is the set of all ground (both base and referential) literals constructible from the predicate symbols of the language and the constants occurring in the program. An *interpretation (for $\mathcal{P}$)* is any subset I of $B_{\mathcal{P}}$. An interpretation is *consistent* if no conflicting literals occur in it. Given an interpretation I, a ground (either simple or referential) literal L is *true* (resp. *false*) *wrt* I if $L \in I$ (resp. $\neg.L \in I$). Let r be a ground rule and I an interpretation; the body of r is *true wrt* I if every A_i, $1 \leq i \leq n$, is true wrt I and every B_j, $1 \leq j \leq m$, is not true w.r.t. I. The rule r is *true wrt* I if either the head of r is true wrt I or its body is not true wrt I. We denote by $\mathcal{P}_g$ the set of all ground rules obtained from each rule $\langle o, r\rangle \in \mathcal{P}$ by replacing each variable appearing in r by each constant appearing in the program. Further, given an object $o \in O$ and a rule r, let $o.r$ denote the rule obtained from r by replacing each simple literal L by $o.L$. Next we define the notion of ground version of $\mathcal{P}$ where authorization propagation along hierarchies is made explicit.

Definition 4. (Ground Version of an Authorization Program) Let $\mathcal{P}_g^* = \bigcup_{o \in O}\{\langle\bar{o}, o.r\rangle \mid \bar{o} \preceq_O o \textit{ and } \langle\bar{o}, r\rangle \in \mathcal{P}_g\}$(the closure of $\mathcal{P}_g$ w.r.t. the object hierarchy). The *ground version* of $\mathcal{P}$, denoted $G(\mathcal{P})$, is inductively defined as follows:
- *Base*: if $\langle\bar{o}, o.r\rangle \in \mathcal{P}_g^*$ then $\langle\bar{o}, o.r\rangle \in G(\mathcal{P})$;
- *Induction*: if $\langle\bar{o}, o.r\rangle \in G(\mathcal{P})$, where r is an authorization rule, then:

1. if $Head(r)$ is of the form $auth(s,p,g)$, then $G(\mathcal{P}) = G(\mathcal{P}) \cup \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$, where:
   - $\Sigma_1 = \{\langle\bar{o}, o.r'\rangle \mid r' = auth(s_1,p,g) \leftarrow auth(s,p,g),\ s \in G \textit{ and } (s_1 \in User\_Groups(s) \cup Role\_Groups(s) \textit{ or } s \preceq_G s_1)\}$, that is, a positive authorization granted to a group propagates to all subgroups, users and roles members of that group.
   - $\Sigma_2 = \{\langle\bar{o}, o.r'\rangle \mid r' = auth(s_1,p,g) \leftarrow auth(s,p,g),\ s \in R \textit{ and } s_1 \prec_R s\}$, that is, a positive authorization granted to a role propagates to all the roles preceding it in the role hierarchy.
   - $\Sigma_3 = \{\langle\bar{o}, o.r'\rangle \mid r' = auth(s,p_1,g) \leftarrow auth(s,p,g), p \prec_P p_1\}$, that is, a positive authorization for a privilege p implies a positive authorization for all the privileges following p in the privilege hierarchy.
2. if $Head(r)$ is of the form $\neg auth(s,p,g)$, then $G(\mathcal{P}) = G(\mathcal{P}) \cup \Theta_1 \cup \Theta_2 \cup \Theta_3$, where:

- $\Theta_1 = \{\langle \bar{o}, o.r' \rangle | r' = \neg auth(s_1, p, g) \leftarrow \neg auth(s, p, g),\ s \in G$ and $(s_1 \in User\_Groups(s) \cup Role\_Groups(s)$ *or* $s \preceq_G s_1)\}$,that is, a negative authorization granted to a group propagates to all subgroups, users and roles members of that group.
- $\Theta_2 = \{\langle \bar{o}, o.r' \rangle | r' = \neg auth(s_1, p, g) \leftarrow \neg auth(s, p, g), s \in R$ *and* $s \prec_R s_1)\}$, that is, a negative authorization granted to a role propagates to all the roles following it in the role hierarchy.
- $\Theta_3 = \{\langle \bar{o}, o.r' \rangle | r' = \neg auth(s, p_1, g) \leftarrow \neg auth(s, p, g), p_1 \prec_P p\}$, that is, a negative authorization for privilege p implies a negative authorization for all the privileges preceding p in the privilege hierarchy.

Thus, $G(\mathcal{P})$ is a set of elements of the form $\langle \bar{o}, o.r \rangle$, that we call *referential rules*. In $\langle \bar{o}, o.r \rangle$, $\bar{o}$ is the *source* object and o is the *referenced* object. A referential rule $\langle \bar{o}, o.r \rangle$ defines an authorization for the (referenced) object o which is inherited from the (source) object $\bar{o}$. The usefulness of the notion of source object will be clarified later on when our conflict resolution policy is formally introduced.

A referential rule $\langle \bar{o}, o.r \rangle \in G(\mathcal{P})$ is true wrt an interpretation I if $o.r$ is true wrt I. We note that in the ground version of a program conflicts may appear. Formally, conflicting rules are defined as follows:

Definition 5. (Conflicting Rules) Two referential rules $r_1 = \langle o_1, o.r \rangle$ and $r_2 = \langle o_2, o'.r' \rangle$ in $G(\mathcal{P})$ are *conflicting* if both the following conditions hold: 1) $o = o'$ (i.e., they refer to the same object); 2) $Head(r)$ and $Head(r')$ are conflicting literals (see subsection 3.1)

We next define our conflict resolution policy.

Definition 6. (Defeating) Let I be an interpretation for $\mathcal{P}$ and let $r_1 = \langle o_1, o.r \rangle$
$\in G(\mathcal{P})$ be a referential rule. We say that r_1 is *defeated* in I if there exists $r_2 = \langle o_2, o'.r' \rangle \in G(\mathcal{P})$ such that r_1 and r_2 are conflicting rules, the body of r_2 is true in I and (at least) one of the following conditions holds (next g and g' are the grantors appearing in $Head(r)$ and $Head(r')$, respectively):

- $g' \prec_R g$, that is, g' is a grantor stronger than g (recall that either $g' = \top_R$, thus r_2 defines a strong authorization, or g and g' are both roles);
- g and g' are incomparable w.r.t. $\prec_R$ (i.e., neither $g' \prec_R g$ nor $g \prec_R g'$) and $o_1 \prec_O o_2$ (thus, the source object of r_2 is a part of the source object of r_1);
- g and g' are incomparable w.r.t. $\prec_R$, o_1 and o_2 are incomparable w.r.t. $\prec_O$ and $Head(r')$ is a negative literal.

Now we are ready to provide the definition of model of an authorization program.

Definition 7. (Authorization Program Model) A *model* for $\mathcal{P}$ is an interpretation M such that every referential rule in $G(\mathcal{P})$ is either true in M or defeated in M.

Proposition 31 *Let M be a model of $\mathcal{P}$. Then, M is a consistent interpretation.*

To define stable models, we need the following preliminary definition.

Definition 8. (Reduction of an Authorization Program) Given an interpretation I for $\mathcal{P}$, the *reduction* of $\mathcal{P}$ w.r.t. I, denoted $G_I(\mathcal{P})$, is the following set of ground rules: $G_I(\mathcal{P}) = \{o.r | \exists \langle \bar{o}, o.r \rangle \in G(\mathcal{P})$ *not defeated in I and Body*(r) *is true w.r.t. I*$\}$.

Note that $G_I(\mathcal{P})$ can be regarded as a Datalog program (with negation by failure) [14, 21] simply by considering each referential literal of the form, say, $o.p(t_1, ..., t_n)$ (resp. $o.\neg p(t_1, ..., t_n)$), as a simple literal with predicate symbol $o.p$ (resp. $o.\neg p$). Thus, given a set X of rules, the immediate consequence operator T_X is defined in the usual way:

$$\begin{aligned} T_X &: 2^{B_\mathcal{P}} \rightarrow 2^{B_\mathcal{P}} \\ I &\mapsto \{L \in B_\mathcal{P} | \ \exists r \in X \ s.t. \ Head(r) = L, \ and \ Body(r) \ is \ true \ wrt \ I\}. \end{aligned}$$

Definition 9. (Stable Model) Let M be a model of $\mathcal{P}$. M is a *stable model* of $\mathcal{P}$ if $M = T^{\infty}_{(G_M(\mathcal{P}))}(\emptyset)$.

Example 9. Consider the program $\mathcal{P}$ of Example 8 and let:

$M_1 = \{$`Employee_Evaluation.`$auth$`(Administrative Manager,write,Top Manager), Program_Repository.`$auth$`(Bob,read,Ann)`$\}$

be an interpretation for $\mathcal{P}$. $G_{M_1}(\mathcal{P}) = \{r_1, r_3\}$. Clearly, $M_1 = T^{\infty}_{(G_{M_1}(\mathcal{P}))}(\emptyset)$ and, thus, M_1 is a stable model of $\mathcal{P}$. It can be easily shown that:

$M_2 = \{$`Employee_Evaluation.`$auth$`(Technical Manager,write,Top Manager), Employee_Salary_Info.`$auth$`(Accountant,write,Top Manager), Program_Repository.`$auth$`(Bob,read,Ann)`$\}$

is also a stable model of $\mathcal{P}$.

Thus, in general, a program is assigned with a number (possibly zero) of stable models. Each stable model represents a consistent way of assigning access authorizations to users.

Example 10. Consider Example 9 above. According to the two stable models associated with $\mathcal{P}$, it is possible to simultaneously authorize `Administrative Manager` to `write` `Employee_Evaluation` and `Bob` to `read` `Program_Repository` and the same happens if we consider `Technical Manager` instead of `Admnistrative Manager`. By contrast, it is not possible to simultaneously authorize `Administrative Manager` to `write` `Employee_Evaluation` and `Accountant` to `write` `Employee_Salary_Info`.

4 Access Control

An access request is represented as a triple (s, p, o), with $s \in U \cup R$, $p \in P$, and $o \in O$. Tuple (s, p, o) states that subject s requires to exercise privilege p on object o.

Let $\Gamma_{\mathcal{P}}$ be the set of stable models of $\mathcal{P}$. Each model belonging to $\Gamma_{\mathcal{P}}$ represents a consistent way of assigning access authorizations. The problem then arises of what model to select, among those in $\Gamma_{\mathcal{P}}$, in order to verify whether an access request can be authorized.

In our access control policy, the choice of the model against which access requests must be verified is driven by the privileges the subjects are exercising on the objects in the system when the access request is issued. Given an access request α, the access control mechanism selects from $\Gamma_{\mathcal{P}}$ a stable model which: *i)* satisfies α and *ii)* is compatible with all the current accesses. If such a model exists, the access is authorized; otherwise, it is denied. Clearly, enforcing this policy requires the system to maintain information on the set of accesses that are currently in execution.

An algorithm enforcing access control is illustrated in Figure 3. The algorithm makes use of two sets, namely *Current_Access* and *Current_Model*. *Current_Access* is the set of current accesses: $(s, p, o) \in Current\_Access$ if and only if subject s is currently exercising privilege p on object o. *Current_Model* is the model of $\mathcal{P}$ currently chosen as valid. Moreover we make use of function Π_g that, given a model M, returns the set of access requests that can be authorized according to that model. Formally, $\Pi_g(M) = \{(s, p, o) \mid o.auth(s, p, g) \in M\}$.

Algorithm 41 **Access Control Algorithm**

```
INPUT:    1) An access request $(s,p,o)$, 2) An authorization program $\mathcal{P}$
OUTPUT:   1) $AUTHORIZE$ if the access request can be authorized,
          2) REJECT, otherwise.
METHOD:

 1. Let $Current\_Access$ be the set of current accesses
 2. if $(s,p,o) \in \Pi_g(Current\_Model)$ then
      $Current\_Access := Current\_Access \cup \{(s,p,o)\}$
      return $AUTHORIZE$
    endif
 3. if $\exists M \in \Gamma_{\mathcal{P}}$ such that $(\{(s,p,o)\} \cup Current\_Access) \subseteq \Pi_g(M)$ then
      $Current\_Model := M$
      $Current\_Access := Current\_Access \cup \{(s,p,o)\}$
      return $AUTHORIZE$
    endif
 4. return $REJECT$
```

Fig. 3. Access Control Algorithm

Example 11. Consider the program $\mathcal{P}$ of Example 9. Assume that *Current_Model* $= M_1$ and *Current_Access* $=$ {`(Bob,read,Program_Repository)`}. Let

(Technical Manager,write,Employee_Evaluation) be a new access request. (Technical Manager,write,Employee_Evaluation) $\notin \Pi_g(M_1)$. However, (Technical Manager,write,Employee_Evaluation) $\in \Pi_g(M_2)$ and *Current_Access* $\subseteq \Pi_g(M_2)$, then the request is authorized and inserted into *Current_Access*. *Current_Model* is set equal to M_2. Suppose now that the access request (Accountant,write,Employee_Salary_Info) is submitted to the system. This request is denied since there does not exist a stable model M of $\mathcal{P}$ such that (Accountant,write,Employee_Salary_Info) $\in \Pi_g(M)$ and *Current_Access* $\subseteq \Pi_g(M)$.

A few concluding remarks on the complexity of the described algorithm. It is well known in logic programming that computing stable model semantics is an exponential task (unless P=NP) [16]. In particular, deciding whether a program admits a stable model is NP-complete. In principle, this is not a drawback, as it means that it is possible to express some NP-complete problems. Without any form of syntactic restrictions (no stratified negation [21], no limitation on the structure of both hierarchies and conflicts, etc.), the language offers a general tool whereby the user is free to model very complex application domains. On the other hand, a number of efficient techniques have recently been proposed to drastically improve the computation of stable models [20, 7, 17, 8]. As a result, the approach becomes feasible for many problems of practical interest. We argue that (real) security applications fall into this category of problems, as it is reasonable to assume that the sources of complexity (i.e., unsolvable conflicts, unstratified negation by failure [21]) are, in general, limited.

5 Conclusions

In this paper we have presented an advanced authorization mechanism based on a logic formalism. The model provides several notable features such as positive/negative and weak/strong authorizations, derivation and propagation rules, and the support for both roles and groups. Moreover, we have proposed an innovative approach for both access control and authorization conflicts resolution.

We plan to extend this work along several directions. First we are developing more articulated resolution policies, taking into account other aspects, such as the group hierarchy. We also plan to extend the current model to incorporate temporal features [2, 3] and to allow a user to take on multiple roles in a section. A third direction concerns implementation issues.

References

1. Abadi, M., Burrows, M., Lampson, B.W., Plotkin, G. A Calculus for Access Control in Distributed Systems. *ACM Trans. on Programming Languages and Systems*, 15(4):706–734, 1993.
2. Bertino, E., Bettini, C., Ferrari, E., Samarati, P. A Temporal Access Control Mechanism for Database Systems. *IEEE TKDE*, 8(1):67–80, 1996.

3. Bertino, E., Bettini, C., Ferrari, E., Samarati, P. An Access Control Mechanism Supporting Periodicity Constraints and Temporal Reasoning. *ACM TODS*, to appear.
4. Bertino, E., Jajodia, S., Samarati, P. Supporting Multiple Access Control Policies in Database Systems. *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland (CA), 1996.
5. Buccafurri, F., Leone, N., Rullo, P. Stable Models and their Computation for Logic Programming with Inheritance and True Negation. *Journal of Logic Programming*, 27(1):5–43, 1996.
6. Buccafurri, F., Leone, N., Scarcello, F. On the Expressive Power of Ordered Logic. *AI Communications*, 9:14-13, 1996.
7. W. Chen, D.S. Warren. Computing of Stable Models and its Integration with Logical Query Processing. *IEEE TKDE*, 17:279-300, 1995.
8. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F., A Deductive System for Nonmonotonic Reasoning, *Proc. of the 4th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, LNAI 1265, Berlin, 1997.
9. E. Fernandez, E.B. Gudes and H. Song. A Model for Evaluation and Administration of Security in Object-Oriented Databases. *IEEE TKDE*, 6:275–292, 1994.
10. Gelfond, M., Lifschitz, V. The Stable Model Semantics for Logic Programming. *Proc. 5th Int. Conf. on Logic Programming*, pp. 1070-1080, 1988.
11. Jajodia, S., Samarati, P., Subrahmanian, V.S., Bertino, E. A Unified Framework for Enforcing Multiple Access Control Policies. *Proc. of ACM-SIGMOD*, 1997.
12. Jajodia, S., Samarati, P. Subrahmanian, V.S. A Logical Language for Expressing Authorizations. *Proc. IEEE Symposium on Research in Security and Privacy*, Oakland (CA), pp. 31–42, 1997.
13. Laenens, E., Saccá, D., Vermeir, D. Extending Logic Programming. In *Proc. of ACM-SIGMOD*, 1990.
14. Lifschitz, V. On the Declarative Semantics of Logic Programs with Negation. *Foundation of Deductive Database and Logic Programming*, pp. 89-148, 1997.
15. Lloyd, J. W. *Foundations of Logic Programming*, Springer-Verlag, 1987.
16. Marek, W., Truszczyński, M., Computing Intersection of Autoepistemic expansions, *Proc. of the 1st Int. Workshop on Logic Programming and Non Monotonic Reasoning*, pp. 37-50, 191.
17. Niemelä, I., Simons, P., Efficient Implementation of the Well-founded and Stable Model Semantics. *Proc. of the 1996 Joint Int. Conf. and Symposium on Logic Programming*, pp. 289–303, Bonn, Germany, 1996.
18. Oracle Corporation. *Oracle8 Server Concepts*, 1997.
19. Rabitti, F., Bertino, E., Kim, E., Woelk, D. A Model of Authorization for Next-Generation Database Systems. *ACM TODS*, 16(1):88–131, 1991.
20. Subrahmanian, V.S., Nau, D. and Vago, C. WFS + Branch and Bound = Stable Models. *IEEE TKDE*, 7(3):362-377, 1995.
21. Ullman, J.D. *Principles of Database and Knowledge-Base Systems, Vol. 1 and 2*, Computer Science Press, 1989.
22. Woo, T.Y.C, Lam, S.S. Authorizations in Distributed Systems: A New Approach. *Journal of Computer Security*, 2(2 & 3):107–136, 1993.