Flexible Support for Multiple Access Control Policies

SUSHIL JAJODIA
George Mason University
PIERANGELA SAMARATI
Università di Milano
MARIA LUISA SAPINO
Università di Torino
and
V. S. SUBRAHMANIAN
University of Maryland

Although several access control policies can be devised for controlling access to information, all existing authorization models, and the corresponding enforcement mechanisms, are based on a specific policy (usually the *closed* policy). As a consequence, although different policy choices are possible in theory, in practice only a specific policy can actually be applied within a given system. In this paper, we present a unified framework that can enforce multiple access control policies within

The work of S. Jajodia was partially supported by the Army Research Office under the contract DAAG-55-98-1-0302; by a DARPA grant administered by a Air Force Research Laboratory, Rome under the contract F30602-98-1-0055; and by the National Science Foundation (NSF) under grant IRI-9622154.
The work of P. Samarati was partially supported by the European Community within the Fifth (EC) Framework Programme under contract IST-1199-11791 – FASTER project.
The work of M. L. Sapino was partially supported by the Italian MURST within the project Intelligent Agent – Interaction and Knowledge Acquisition.
The work of V. S. Subrahmanian was supported in part by the Army Research Office under grants DAAG-55-98-1-0302, DAAH-04-95-10174, DAAH-04-96-10297, DAAG-55-97-10047, and DAAH-04-96-1-0398; by the Army Research Laboratory under contract number DAAL-01-97-K0135; by DARPA/Rome Lab contract F306029910552; and by an NSF Young Investigator award IRI-93-57756.
Authors' addresses: S. Jajodia, Center for Secure Information Systems, George Mason University, Fairfax, VA 22030-4444, e-mail: jajodia@gmu.edu; P. Samarati, Dipartimento di Technologie dell'Informazione, Università di Milano, 26013 Crema, Italy, e-mail: samarati@dsi.unimi.it; M. L. Sapino, Dipartmento di Informatica, Università di Torino, 10149 Torino, Italy, e-mail: mlsapino@di.unito.it; V. S. Subrahmanian, Department of Computer Science, University of Maryland, College Park, MD 20742, e-mail: vs@cs.umd.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1(212) 869-0481, or permissions@acm.org.
© 2001 ACM 0362-5915/01/0600–0214 $5.00

a single system. The framework is based on a language through which users can specify security policies to be enforced on specific accesses. The language allows the specification of both positive and negative authorizations and incorporates notions of authorization derivation, conflict resolution, and decision strategies. Different strategies may be applied to different users, groups, objects, or roles, based on the needs of the security policy. The overall result is a flexible and powerful, yet simple, framework that can easily capture many of the traditional access control policies as well as protection requirements that exist in real-world applications, but are seldom supported by existing systems. The major advantage of our approach is that it can be used to specify different access control policies that can all coexist in the same system and be enforced by the same security server.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration–*security, integrity, and protection*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Management, Security

Additional Key Words and Phrases: Access control policy, authorization, logic programming

1. INTRODUCTION

With the advent of the information superhighway, there is now an immense amount of information available in a wide variety of information sources. A database administrator (DBA) or system security officer (SSO) may be responsible for providing secure, authorized access to a collection of distributed objects that includes some files, some relations, some object bases, some images, etc. The SSO requires an authorization model that can be used to restrict access to different classes of data objects (files, relations, objects, images, etc.). Moreover, the SSO may wish to use one access control policy to regulate access to the image data, another to regulate access to the relational data, and yet a third policy to regulate access to the object-oriented data. This situation requires databases, operating systems, and file systems (and, in general, systems that create and manipulate objects) that provide a way for the SSO to apply different access control policies to different classes of data objects.

Several access control policies have been proposed in the literature to govern access to information by users. Correspondingly, several authorization models have been formalized and access control mechanisms enforcing them implemented [Castano et al. 1995]. Each model, and its corresponding enforcement mechanism, implements a single specified policy, which is in fact built into the mechanism. As a consequence, although different policy choices are possible in theory, each access control system is in practice bound to a specific policy. The major drawback of this approach is that a single policy simply cannot capture all the protection requirements that may arise over time. When a system imposes a specific policy on users, they have to work within the confines imposed by the policy. When the protection requirements of an application are different from the policy built into the system, in most cases, the only solution is to implement the policy as part of the application code. This solution, however, is dangerous from a security viewpoint since it makes the tasks of verification, modification, and adequate enforcement of the policy difficult.

Limitations of existing access control systems, which are generally based on the closed policy, have been pointed out by other researchers [Lunt 1989].

To overcome such limitations, more recent authorization models also permit specification of negative authorizations stating accesses to be denied [Bertino et al. 1993; 1999; Brüggemann 1992; Denning et al. 1987; Rabitti et al. 1991]. Possible conflicts arising from the presence of both negative and positive authorizations are resolved through a specific policy built into the model. Examples of conflict resolution policies that have been applied in different proposals are: denials-take-precedence [Bertino et al. 1993], most-specific takes precedence [Denning et al. 1987], most-specific together with the concept of strong and weak authorizations [Bertino et al. 1999; Rabitti et al. 1991], explicit specification of priorities [Shen and Dewan 1992], and explicit specification of the policy to be applied [Jonscher and Dittrich 1996]. Although these authorization models allow a flexible and easy specification of authorizations, they remain limited in the access control policies they can express as they rely on the particular properties of the underlying data model (generally, relational or object-oriented data model). As a result, these authorization models cannot be easily extended to other data models. In particular, it is not clear how these models can be used to restrict access in the current World Wide Web (WWW) environment where a wide variety of information sources (files, relations, objects, images, etc.) exist.

In this paper, we define a *Flexible Authorization Framework* (FAF) that allows the specification of accesses to be allowed or denied in an expressively powerful, declarative, and flexible manner. The framework is based on a language through which users can specify security policies to be enforced on specific accesses. The language allows the specification of both positive and negative authorizations and incorporates notions of authorization derivation, conflict resolution, and decision strategies. Such strategies can exploit the hierarchical structures in which system components (objects, users, groups, and roles) are organized as well as any other relationship that the SSO may wish to exploit, in a flexible way. The language permits the specification of general constraints on authorizations as well as on the derivation and conflict resolution process. The overall result is a flexible and powerful, yet simple, framework that allows us to easily capture many traditional access control policies as well as many protection requirements existing in real world applications that could not, or could only partially be enforced by previous approaches. We impose restrictions on the logic rules considered by the framework that, while not limiting expressiveness, ensure the implementability of our approach and therefore its suitability to real world applications.

The rest of the paper is organized as follows. Section 2 introduces and formally defines the basic components of our authorization framework and of the data system to which protection must be ensured. Section 3 formally defines the object and subject hierarchies within the data system and introduces the concept of authorization specification. Sections 3.2 through 3.4 describe several examples of propagation, conflict resolution, and decision policies found in the literature. Section 4 presents the language on which our framework is based and illustrates how it can capture the different policies discussed as well as several protection requirements that may need to be enforced. Section 5 discusses implementation issues and presents a materialization technique to compute and explicitly maintain all accesses to be allowed or denied, for

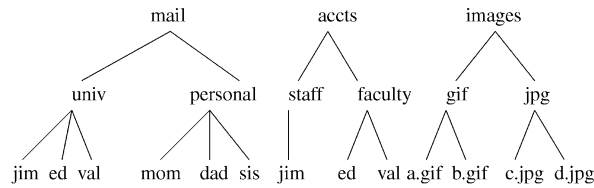

Fig. 1. Organization of an example directory.

efficient access control enforcement. It also presents an approach for updating the materialization upon changes to the specifications. Section 6 compares our work with previous related work. Section 7 concludes the paper.

2. COMPONENTS OF A FLEXIBLE AUTHORIZATION FRAMEWORK (FAF)

In this section, we first provide an intuitive overview of the components of an authorization framework (Section 2.1). Then, we provide a simple basic set of definitions that allow us to model these components (Section 2.2) and show how these definitions can capture the basic components as instances of these general definitions.

2.1 An Intuitive Description of FAF Components

Any authorization framework must provide a means to express answers to the following questions:

(1) **To what data items is the framework mediating access and how are these data items organized?**
(2) **For what kinds of accesses does the framework determine authorization privileges?**
(3) **How are the users/user groups to which the framework considers granting access organized?**
(4) **What types of roles may users adopt, and under what conditions may they adopt these roles? Do access privileges change as users adopt different roles?**
(5) **Who can administer (e.g., grant and revoke) accesses?**

2.1.1 *Data Items.* In general, any authorization framework determines the circumstances under which a user may attempt to execute an access operation on a given data item. However, in most realistic systems, data items are organized hierarchically. For example, in a file system, the basic objects are files, but these files are typically organized in a hierarchical directory structure. Similarly, in an object oriented database, the objects being accessed are organized into an object hierarchy. When specifying an authorization policy using a FAF, we would like to specify policies that apply to arbitrary directories (e.g. "All files contained in the `payroll` directory are accessible only to individuals in the `accounting` department") or to arbitrary classes of objects (e.g. "All MPEG files are accessible only to `vice-presidents` or higher"). Figure 1 shows a simple directory structure, while Figure 2 shows a simple object hierarchy. In both

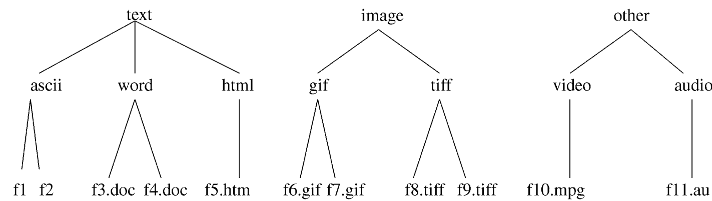

Fig. 2. Organization of an example OO hierarchy.

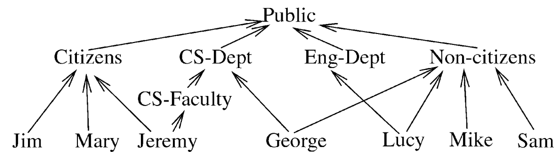

Fig. 3. An example user-group hierarchy.

cases, the reader will observe that the data items in question are partially ordered.

2.1.2 *Access Types.* We assume that there exists some set of *actions* or operations that the user tries to execute on different data objects. In the case of file systems, these actions may include `read`, `write`, `move`, and `copy` operations. In the case of object hierarchies, these operations may involve invocation of different methods.

2.1.3 *Users and User Groups.* In general, when describing authorizations, we assume that there is some set of users, as well as groups consisting of such users. The word "user" always refers to a human being, while a group is a nonempty set of users. In most applications, users and groups are organized into a hierarchy–this hierarchy typically looks like a directed acyclic graph such that the nodes of in-degree zero correspond to the users. Figure 3 shows a simple hierarchy of users and groups.

2.1.4 *Roles.* It is very common for users to assume roles. For example, Jane might be a professor in the Computer Science Department. When the department chair goes on leave, Jane may assume the role of department chair. In such a situation, Jane is playing a role that may bestow upon her privileges that she did not have before. These privileges apply only while Jane is playing this role–when she stops playing the role, she no longer has the ability to execute the privileges associated with the role. Roles too may be organized as a hierarchy–for example, roles such as `chair`, `dean`, etc. all form a hierarchy as shown in Figure 4.

2.1.5 *Administration.* Any FAF must include an administrative policy that regulates who can grant authorizations and revoke them. In this paper, we consider a centralized administration policy, where authorization administration is the task of a single administrator, whom we refer to as *system security officer* (*SSO*).

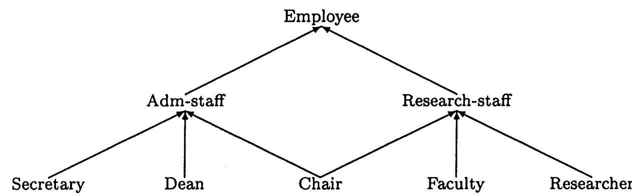

Fig. 4. An example role hierarchy.

2.2 Formal Definitions

In all cases thus far, we have noticed that data items, actions, users/groups, and roles may be organized as hierarchies. In each of these cases, there are certain primitive entities (e.g., files) and certain aggregate entities (e.g., directories) that consist of sets of primitive entities.

2.2.1 *Hierarchies.* It will turn out that a single mathematical structure called a *hierarchy* (to be defined below) is rich enough to capture the structure of data items, users/groups, and roles.

Definition 2.1 (*Hierarchy*). A *hierarchy* is a triple $(\mathsf{X}, \mathsf{Y}, \leq)$ where:

(1) X and Y are disjoint sets

(2) $\leq$ is a partial order on $(\mathsf{X} \cup \mathsf{Y})$ such that each $x \in \mathsf{X}$ is a minimal element of $(\mathsf{X} \cup \mathsf{Y})$; an element $x \in \mathsf{X}$ is said to be minimal iff there are no elements below it in the hierarchy, that is iff $\forall y \in (\mathsf{X} \cup \mathsf{Y})$: $y \leq x \Rightarrow y = x$.

In this definition, X may be thought of as an abstraction of the primitive entities, and Y is an abstraction of the set of aggregate entities. For example:

– We may capture data items and groups of data items via the triple $\mathsf{OTH} = (\mathsf{Obj}, \mathsf{T}, \leq_{\mathsf{OT}})$ where Obj is a set of identifiers of objects in a database, T is a set of *types* (or sets of objects), and $\leq_{\mathsf{OT}}$ is a partial ordering such that for all $x \in \mathsf{Obj}$ and $\tau \in \mathsf{T}$, $x \leq_{\mathsf{OT}} \tau$ iff x is of type τ.

– Similarly, we may capture the idea of users and groups via a hierarchy $\mathsf{UGH} = (\mathsf{U}, \mathsf{G}, \leq_{\mathsf{UG}})$ where U is a set of user-ids, and G is a set of identifiers corresponding to named collections of users. $\leq_{\mathsf{UG}}$ is a partial ordering such that for all users $u \in \mathsf{U}$ and all groups $\gamma \in \mathsf{G}$, $u \leq_{\mathsf{UG}} \gamma$ iff user u is in group γ.

– In the same way, roles also may be captured as a hierarchy $\mathsf{RH} = (\emptyset, \mathsf{R}, \leq_{\mathsf{R}})$ where R is a set of roles and $x \leq_{\mathsf{R}} y$ if x is a specialization of y. A role is a specialization of another role if it refers to more specialized activities (which generally implies carrying more privileges). For instance, `Faculty` and `Researcher` can be both seen as a specialization of `Research-staff`. Note that there is no such thing as a primitive role and that users (who are minimal elements of the user-group hierarchy), do not appear in the role hierarchy. The association between users and roles is accomplished through explicit activation, regulated by authorizations (see Section 4).

Note the distinction we are making between users, groups, and roles. Users are people connecting to the system; groups are *named* sets of users; and

roles are *named* collections of privileges needed to perform specific activities in the system. There are several differences between groups and roles:

– Groups define a grouping of *people* while roles define grouping of *privileges*.
– Roles can be activated and deactivated by users at their discretion while group membership typically cannot be activated or deactivated by the user [Jajodia et al. 1997a] (though a system manager may periodically modify group membership). For instance, consider UGH and RH in Figures 3 and 4. Suppose `Jeremy` can assume any of the following roles: `Faculty`, `Research-Staff`, and `Employee`. When `Jeremy` accesses the system, he will always be considered to be a member of the following groups: `Citizens`, `CS-Faculty`, and, indirectly, `CS-Dept` and `Public`. Hence, he will always enjoy the privileges available to these groups, as well as be subject to the restrictions imposed on him by virtue of his membership in these groups. Jeremy cannot decide by himself that he does not want to belong to the group `Citizens` at a specific time. On the other hand, Jeremy can activate/deactivate certain roles at his will. In these cases, he will only have the privileges associated with a given role when he is in that role–as soon as he steps out of the role, he loses the privileges associated with that role.

The dynamic aspect of roles has a double advantage. First, it allows the limitation of the use of the privileges needed to execute the tasks associated with the role only within the specific task execution. Second, it allows the enforcement of the *least privilege principle* according to which tasks are granted only those privileges needed to complete their execution. It is important to note that groups and roles are two complementary, not exclusive, concepts. Also, roles and groups are not necessarily disjoint concepts–for instance, a group `G_Faculty` can be defined to which all users who are faculty members belong, and a role `R_Faculty` can be defined to which privileges specifically related to the professor's activity are granted. Use of privileges granted to the group and to the role will be regulated as described above.

Definition 2.2 (*Disjoint Hierarchies*). Two hierarchies $H_1 = (X_1, Y_1, \leq_1)$ and $H_2 = (X_2, Y_2, \leq_2)$ are *disjoint* iff $(X_1 \cup Y_1) \cap (X_2 \cup Y_2) = \emptyset$.

Intuitively, two hierarchies are disjoint iff they share no common elements. For example, the hierarchies of Figure 1 and 2 are not disjoint, because they share the common element "gif". In contrast, the hierarchies in Figures 3 and 4 are disjoint.

Notational Abuse. Throughout this paper, we will often think of a hierarchy $H = (X, Y, \leq)$ as the set $X \cup Y$, and talk of H's members as the members of $X \cup Y$.

2.2.2 *Data System, Formalized.* In previous sections, we have informally spoken about "the system" or a "data system." In general, a data system consists of users/groups, the data they are accessing, together with the roles they may play, and the types of access modes they use. Building upon our individual

definitions of these basic concepts, we are now ready to formally define a data system.

Definition 2.3 (*Data System*). A *Data System* **D***S* is a 5-tuple (OTH, UGH, RH, A, Rel) where:

(1) OTH = (Obj, T, $\leq_{\mathsf{OT}}$) is an object-type hierarchy;
(2) UGH = (U, G, $\leq_{\mathsf{UG}}$) is a user-group hierarchy;
(3) RH is a role hierarchy RH = (∅, R, $\leq_{\mathsf{R}}$);
(4) A is a set whose elements are called *authorization modes* or *actions* (we will use the notation SA to denote the set $\{+a, -a \mid a \in \mathsf{A}\}$);
(5) Rel is a set whose elements are called *relationships*. Relationships can be defined on the different elements of **DS** and may be unary, binary or *n*-ary in nature.

Furthermore, OTH, UGH, and RH are disjoint.

Note that the above definition is very general. Most individual data systems we encounter are instantiations of this general definition, by assigning appropriate entities to each of the five components listed above. Different assignments lead to different data system instances.

For example, we may consider ordinary file systems to be a data system instance by taking OTH to be a directory hierarchy such as the one shown in Figure 1, UGH to be a user-group hierarchy such as the one shown in Figure 3, and RH to be the role hierarchy shown in Figure 4. Here, A could include actions such as `r`, `w`, `x` (read, write, execute). Similarly, relationships may include entites such as `subordinate` specifying a binary relationship between users, `owner` specifying a binary relationship between users and objects, `readers` specifying a binary relationship between users and files, showing that certain readers have read certain files, etc. The reader will notice that relational databases and object-oriented systems may be very easily extended to be instances of data systems as well.

In addition, the notion of data system we have defined is hierarchical since it assumes that a hierarchy is defined on objects and types, users and groups, and roles, respectively. This is not a limitation: a data system with no object-type, user-group, or role hierarchy can be captured as a hierarchical data system. In particular, a data system with no object-type hierarchy is represented by a data system with $\mathsf{T} = \emptyset$ and where $x \leq_{\mathsf{OT}} y$ iff $x = y$. A data system with no user-group hierarchy is represented by a data system with $\mathsf{G} = \emptyset$ and where $x \leq_{\mathsf{UG}} y$ iff $x = y$. A data system with no role hierarchy is represented by a data system where $x \leq_{\mathsf{R}} y$ iff $x = y$. Note also that a data system which does not support the concept of roles can be represented by a data system where $\mathsf{R} = \emptyset$. Thus, hierarchical data systems are more general than data systems that lack hierarchy. Given this, in the rest of this paper, we deal with object-type, user-group, and role-hierarchical data systems.

At this stage, we have completed our definition of the entities that must be taken into account when defining a flexible authorization framework. We are now ready to describe our actual authorization model.

3. AUTHORIZATION SPECIFICATION AND POLICIES

Throughout this section, we will assume that we are referring to some arbitrary, but fixed data system, **DS** = (OTH, UGH, RH, A, Rel).

3.1 Authorizations

In this section, we use the expression *authorization subject* to denote those entities *for* which authorization privileges can be specified–authorization subjects are therefore users, groups, and roles. Likewise, the expression *authorization object* refers to entities *on* which authorizations can be specified–authorization objects therefore include objects, types, and roles. Roles are included in authorization objects as they can be activated/deactivated by authorization subjects.

We use AO, AS, SA to denote set of authorization objects, authorization subjects, and signed actions (see below), respectively. When it is clear from context, we will often merely write “subject” and “object,” rather than “authorization subject” and “authorization object.”

Definition 3.1 (*Authorization Subject Hierarchy*). Let **DS** = (OTH, UGH, RH, A, Rel) be a data system. The authorization subject hierarchy associated with **DS** is the hierarchy ASH = (U, G∪R, $\leq_{\mathsf{AS}}$), where $\leq_{\mathsf{AS}}$ is defined as follows:

$$x \leq_{\mathsf{AS}} y \ \text{ iff } \{x, y\} \subseteq \mathsf{U} \cup \mathsf{G} \ \& \ x \leq_{\mathsf{UG}} y \ \text{ or } \\ \{x, y\} \subseteq \mathsf{R} \ \& \ x \leq_{\mathsf{R}} y.$$

Using a graphical intuition, the graph of ASH is obtained by placing the graphs of UGH and RH side by side. Similarly, we may define an authorization object hierarchy as follows:

Definition 3.2 (*Authorization Object Hierarchy*). Let **DS** = (OTH, UGH, RH, A, Rel) be a data system. The authorization object hierarchy associated with **DS** is the hierarchy AOH = (Obj, T ∪ R, $\leq_{\mathsf{AO}}$), where $\leq_{\mathsf{AO}}$ is defined as follows:

$$x \leq_{\mathsf{AO}} y \ \text{ iff } \{x, y\} \subseteq \mathsf{Obj} \cup \mathsf{T} \ \& \ x \leq_{\mathsf{OT}} y \ \text{ or } \\ \{x, y\} \subseteq \mathsf{R} \ \& \ y \leq_{\mathsf{R}} x.$$

Intuitively, the authorization object hierarchy is obtained by placing the OTH and the inverse of RH side by side. The reason why RH is inverted is to simplify the authorization propagation rules for authorization objects. Intuitively, while authorizations to execute an action *on* a type propagate down the object-type hierarchy to its subtypes and objects, authorizations to execute an action *on* a role propagate up the role hierarchy, to its more generic (less privileged) roles. By inverting the role hierarchy in the definition of the authorization object hierarchy, we do not need to worry about this different behavior and can simply propagate authorizations down the authorization object hierarchy, regardless of whether the authorization object is a type or a role.

We are now ready to specify authorizations. Intuitively, an authorization specifies which authorization subjects can (or cannot) perform which actions on which authorization objects.

Definition 3.3 (*Authorization*). An *authorization* is a triple of the form $(o, s, \langle\text{sign}\rangle a)$ where $o \in$ AO, $s \in$ AS, $a \in$ A and "sign" is either "+" or "−".

Informally, the triple $(o, s, +a)$ states that authorization subject s can execute action a on authorization object o. Similarly, the triple $(o, s, -a)$ states that authorization subject s cannot execute action a on authorization object o. A few simple example authorizations are given below.

- `(mail,faculty,+read)`.
  This authorization states that the authorization subject `faculty`, shown in Figure 4, can execute the action `read` on the authorization object `mail`, shown in Figure 1.
- `(personal,faculty,-read)`.
  This authorization states that the authorization subject `faculty`, shown in Figure 4, may not execute the action `read` on the authorization object `personal`, shown in Figure 1.

The above two examples bring us to a significant point–one that will take up much of the rest of this paper. How should authorizations be *propagated* through the authorization subject and object hierarchies? For example, we see above that we might wish to propagate the authorization `(mail, faculty, +read)` to the sub-directory, `univ`, to obtain a *derived* authorization `(univ, faculty, +read)`. This allows the faculty to read files in the `univ` directory.

In the next section, we introduce several example propagation policies. In Sections 3.3 and 3.4, we introduce several conflict resolution and decision policies.

3.2 Example Propagation Policies

In this section, we formally define different propagation policies on a single hierarchy. As usual, we assume the existence of an arbitrary, but fixed data system $\mathbf{DS} = (\text{OTH}, \text{UGH}, \text{RH}, \text{A}, \text{Rel})$. The general idea is as follows. Given a hierarchy H, we would like to label each node in the hierarchy with pairs such that the node label and the two other components of the pair jointly determine a set of authorization triples $(o, s, \pm a)$. Since we would like such labelings to be possible on all the hierarchies we have described thus far in this paper, we now introduce a generic notion of a hierarchy labeling.

Definition 3.4 (*Hierarchy Labeling*). Let AUTH be a set of authorizations and $\text{H} = (\text{X}, \text{Y}, \leq)$ be a hierarchy. Let $\text{X} \cup \text{Y} \subseteq \alpha$ for some $\alpha \in \{\text{AO}, \text{AS}, \text{SA}\}$ and let LABELS be the cartesian product of the sets in $\{\text{AO}, \text{AS}, \text{SA}\} - \{\alpha\}$. A *hierarchy labeling* of hierarchy H wrt AUTH is a partial mapping $\lambda_{\text{H}}^{\text{AUTH}} : \text{X} \cup \text{Y} \to 2^{\text{LABELS}}$ such that: $\forall x \in \text{X} \cup \text{Y} : l \in \lambda_{\text{H}}^{\text{AUTH}}(x) \Leftrightarrow (x, l) \in \text{AUTH}$.[1]

The above definition is very generic, and allows the system security officer to first choose the hierarchy $\text{H} = (\text{X}, \text{Y}, \leq)$ that he wants to work with. For example, he may choose the authorization object hierarchy AOH described earlier.

[1]Provided that the order of the elements in the tuple is rearranged.

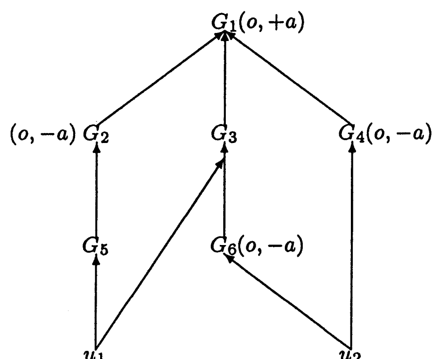

Fig. 5. An example of labeled authorization subject hierarchy.

Once H is selected, LABELS is the Cartesian product of the sets not involved in hierarchy H. In our case, this set involves all pairs $(s, \pm a)$ where s is a subject and $\pm a$ specifies authorization privileges for action a. As our hierarchy H in this example consists of objects, we would like each object in the hierarchy to have an associated set of pairs of the form $\{(s_1, \pm a_1), (s_2, \pm a_2), \ldots, (s_m, \pm a_m)\}$ indicating that a subject s_i is authorized $(+a_i)$ or denied $(-a_i)$ some action a_i.

In contrast, the following example shows a situation where the hierarchy selected by the system security officer is the authorization subject hierarchy.

Example 3.1. Figure 5 illustrates an example of a labeled authorization subject hierarchy where, for the sake of simplicity, we have taken $\mathsf{R} = \emptyset$. In this case, we have $\mathsf{AUTH} = \{(o, G_1, +a), (o, G_2, -a), (o, G_4, -a), (o, G_6, -a)\}$. For example, the label associated with node G_4 says that users in group G_4 may not execute action a on object o.

The reader should note that in the definition of $\lambda_{\mathsf{H}}^{\mathsf{AUTH}}$, it is usually the case that the user does not specify AUTH explicitly. Rather, a graphical user interface is used to pick a hierarchy, and then label it appropriately, causing AUTH to be generated by the interface.

We are now ready to describe various different authorization propagation policies that have been or can be used in a wide variety of real world situations. The reason we need authorization propagation policies is because in most real applications, the system security officer only specifies a partially labeled hierarchy, rather than one that has labels for all the nodes. Propagating authorizations corresponds to extending a partially labeled hierarchy to a more complete labeling (even though the resulting labeling may not be fully complete). A propagation policy is a map π that, given a hierarchy H and an input set AUTH of authorizations, returns as output a set $\mathsf{AUTH}' \supseteq \mathsf{AUTH}$ of authorizations (the new authorizations are "derived" ones). We report below some example policies. Figure 6 reports the result of applying them to the labeled hierarchy in Figure 5.

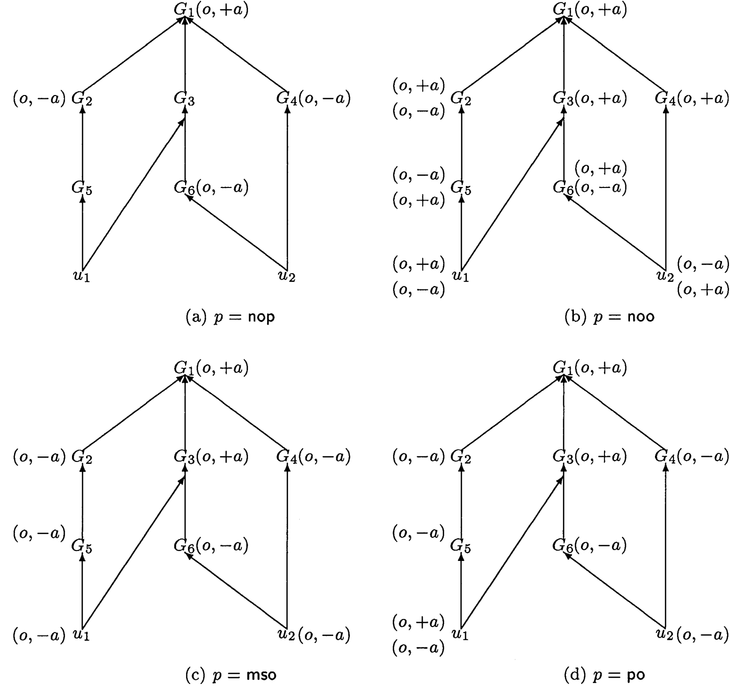

Fig. 6. Derivation of $\mathsf{AUTH}' = \pi^{p}_{\mathsf{H}}(\mathsf{AUTH})$ for H and AUTH of Example 3.1.

No propagation $(\pi_{\mathsf{H}}^{\mathsf{nop}})$[2] Authorizations are not propagated, that is, $\mathsf{AUTH}' = \mathsf{AUTH}$.

$\forall x \in \mathsf{X} \cup \mathsf{Y}:$
$(z, +a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow (z, +a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(x)$
$(z, -a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow (z, -a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(x)$

No overriding $(\pi_{\mathsf{H}}^{\mathsf{noo}})$ All the authorizations of a node are propagated to its subnodes, regardless of the presence of other contradicting authorizations.

$\forall x \in \mathsf{X} \cup \mathsf{Y}:$
$(z, +a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow \exists y : y \in \mathsf{X} \cup \mathsf{Y}, x \leq y, (z, +a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(y)$
$(z, -a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow \exists y : y \in \mathsf{X} \cup \mathsf{Y}, x \leq y, (z, -a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(y)$

Most specific overrides $(\pi_{\mathsf{H}}^{\mathsf{mso}})$ Authorizations of a node are propagated to its subnodes if not overridden. The label attached to a node n overrides a contradicting label of any of its supernodes for all the subnodes of n. For instance, with reference to Figure 5, the negative authorization of G_2 overrides the authorization of G_1 for G_2 and all its members. In particular, only the authorization of G_2 (and not the authorization of G_1) will be propagated to u_1 (see Figure 6).

[2]Note that this policy should not be used with the group hierarchy. Authorizations for groups are meant to apply to the group members and have therefore meaning only if authorizations are propagated.

$$\forall x \in \mathsf{X} \cup \mathsf{Y}:$$
$$(z,+a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow \exists y,\ \nexists w: y, w \in \mathsf{X} \cup \mathsf{Y},\ y \neq w, x \leq y, x \leq w \leq y, (z,+a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(y), (z,-a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(w)$$
$$(z,-a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow \exists y,\ \nexists w: y, w \in \mathsf{X} \cup \mathsf{Y},\ y \neq w, x \leq y, x \leq w \leq y, (z,-a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(y), (z,+a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(w)$$

Path overrides ($\pi_{\mathsf{H}}^{\mathsf{po}}$) Authorizations of a node are propagated to its subnodes if not overridden. The label attached to a node n overrides a contradicting label of a supernode n' for all the subnodes of *n only for the paths passing from n* [Bertino et al. 1999]. The overriding has no effect on other paths. In case n is nonminimal, the authorization of n' may still reach the subnodes of n through other paths. For instance, with reference to Figure 5, the negative authorization of G_2 overrides the authorization of G_1 for G_2 and all its members only for the membership paths passing from G_2. The authorizations of G_1 can possibly still be propagated to the subjects below it in the hierarchy through other membership paths. In particular, the authorization specified for G_1 will be propagated to u_1 through the membership path from u_1 to G_1 passing from G_3 (see Figure 6).

$$\forall x \in \mathsf{X} \cup \mathsf{Y}:$$
$$(z,+a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow \exists y, p,\ \nexists w: y, w \in \mathsf{X} \cup \mathsf{Y},\ y \neq w,\ p \text{ is a path between } x \text{ and } y \text{ in } \mathsf{H}, w \text{ appears in } p, (z,+a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(y), (z,-a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(w)$$
$$(z,-a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}'}(x) \Leftrightarrow \exists y, p,\ \nexists w: y, w \in \mathsf{X} \cup \mathsf{Y},\ y \neq w,\ p \text{ is a a path between } x \text{ and } y \text{ in } \mathsf{H}, w \text{ appears in } p, (z,-a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(y), (z,+a) \in \lambda_{\mathsf{H}}^{\mathsf{AUTH}}(w)$$

It is important for the reader to note that in a data system **DS**, we have three hierarchies. In addition, hybrid hierarchies such as AOH and ASH may be obtained by merging the initial hierarchies present in **DS**. The authorizations specified by the system security officer induce a corresponding labeling of each of these hierarchies. Furthermore, the system security officer may specify different propagation policies for each of the hierarchies. For instance, authorizations can be propagated according to the most specific overrides policy ($\pi_{\mathsf{AS}}^{\mathsf{mso}}$) along the subject hierarchy and not be propagated ($\pi_{\mathsf{OTH}}^{\mathsf{nop}}$) along the object hierarchy. Note also that, within the same hierarchy, disjoint subhierarchies can be identified to which different propagation policies can be applied. For instance, within the subject hierarchy, roles can be subject to a propagation policy (e.g., $\pi_{\mathsf{RH}}^{\mathsf{nop}}$) and groups be subject to another policy ($\pi_{\mathsf{UGH}}^{\mathsf{mso}}$). Such labelings may lead to conflicts. This is the topic of the next section.

3.3 Example Conflict Resolution Policies

As described above, conflicts may arise in authorization specifications for a variety of reasons. In this section, we introduce various example conflict resolution policies. A conflict resolution policy is represented as a mapping, γ, that takes as input a set AUTH of authorizations and returns as output a set of authorizations $\mathsf{AUTH}' \subseteq \mathsf{AUTH}$. Examples of different conflict resolution policies are given below.

No Conflicts (γ^{nc}) This policy assumes that no conflicts occur. The presence of conflicts is considered an error.

$$\forall o,s,a:$$
$$(o,s,+a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,+a) \in \mathsf{AUTH} \wedge (o,s,-a) \notin \mathsf{AUTH}$$
$$(o,s,-a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,-a) \in \mathsf{AUTH} \wedge (o,s,+a) \notin \mathsf{AUTH}$$
$$(o,s,+a) \in \mathsf{AUTH} \wedge (o,s,-a) \in \mathsf{AUTH} \Rightarrow \textit{error}$$

Denials take precedence (γ^{dtp}) In this case, negative authorizations are always adopted when a conflict occurs. In other words, the principle says that if we have one reason to authorize an access, and another to deny it, then we deny it. With respect to the authorizations in Figure 6(b,d), the negative authorization will be considered to hold for user u_1.

$$\forall o,s,a:$$
$$(o,s,+a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,+a) \in \mathsf{AUTH} \wedge (o,s,-a) \notin \mathsf{AUTH}$$
$$(o,s,-a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,-a) \in \mathsf{AUTH}$$

Permissions take precedence (γ^{ptp}) In this case, positive authorizations are always adopted when a conflict occurs. In other words, the principle says that if we have one reason to authorize an access and another to deny it, then we authorize it. With respect to the authorizations in Figure 6(b,d), the positive authorization will be considered to hold for user u_1.

$$\forall o,s,a:$$
$$(o,s,+a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,+a) \in \mathsf{AUTH}$$
$$(o,s,-a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,-a) \in \mathsf{AUTH} \wedge (o,s,+a) \notin \mathsf{AUTH}$$

Nothing takes precedence (γ^{ntp}) This principle says that we neither authorize nor deny an access when there is a conflict. (Instead, we make a decision by deferring to the decision policies that force a decision and that are discussed later in Section 3.4). With respect to the authorizations in Figure 6(b,d), no authorizations will be considered to hold for user u_1.

$$\forall o,s,a:$$
$$(o,s,+a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,+a) \in \mathsf{AUTH} \wedge (o,s,-a) \notin \mathsf{AUTH}$$
$$(o,s,-a) \in \mathsf{AUTH}' \Leftrightarrow (o,s,-a) \in \mathsf{AUTH} \wedge (o,s,+a) \notin \mathsf{AUTH}$$

3.4 Example Decision Policies

Once we have propagated authorizations within a hierarchy and have resolved conflicts, there exists the possibility that some accesses are neither authorized nor denied. In such cases, a firm decision must be taken. Decision policies force accesses to be either authorized or denied. Formally, a decision policy is a function δ: $\mathsf{AO} \times \mathsf{AS} \times \mathsf{A} \rightarrow \{+,-\}$. $\delta(o,s,a) = +$ if s is to be allowed action a on object o. $\delta(o,s,a) = -$ if s is to be denied action a on object o. Two well-known decision policies are described below.

Open $\delta^{\text{open}}(o,s,a) = + \Leftrightarrow (o,s,-a) \notin \mathsf{AUTH}$
$\delta^{\text{open}}(o,s,a) = - \Leftrightarrow (o,s,-a) \in \mathsf{AUTH}$

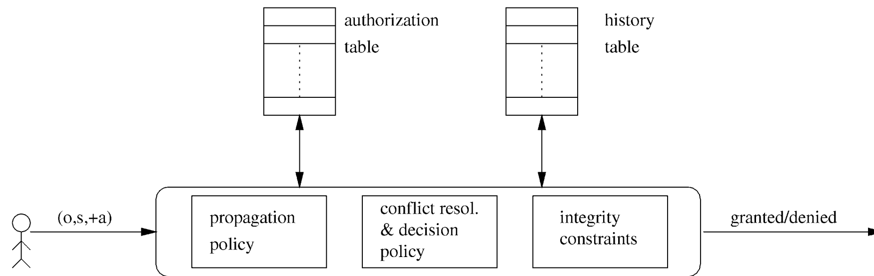

Fig. 7. Functional authorization architecture.

Closed $\delta^{\text{closed}}(o, s, a) = + \Leftrightarrow (o, s, +a) \in \text{AUTH}$
$\delta^{\text{closed}}(o, s, a) = - \Leftrightarrow (o, s, +a) \notin \text{AUTH}$

The open policy denies access if there exists a corresponding negative authorization and allows it otherwise. In contrast, the closed policy allows an access if there exists a corresponding positive authorization and denies it otherwise. To illustrate, consider the sets of authorizations graphically illustrated in Figure 6 and assume that the nothing takes precedence policy is applied for conflict resolution. With the open policy, u_1 would be allowed access a on o in the case of Figure 6(a) and denied in Figure 6(b,c,d). With the closed policy, u_1 would be denied access in Figure 6(a,c) but allowed access in Figure 6(b,d).

Note that decision policies apply to the set of authorizations produced by the propagation and conflict resolution policies and that they force a decision on accesses for which there are no authorizations or there are unresolved conflicts. Note also that, with respect to unresolved conflicts, the open policy is equivalent to a permissions take precedence resolution, while the closed policy is equivalent to a denials take precedence resolution.

4. AUTHORIZATION FRAMEWORK AND SPECIFICATION LANGUAGE

4.1 Architecture of the Authorization Framework

The architecture of our authorization framework (shown in Figure 7) consists of the following components (in addition to the data system being considered):

–A *history table* whose rows describe the accesses executed;

–An *authorization table* whose rows are authorizations composed of the triples described above;

–A *propagation policy* that specifies how to obtain new derived authorizations from the explicit authorization table described above;

–A *conflict resolution and decision policy* that: (i) specifies how to eliminate conflicts that arise when two or more conflicting authorizations apply to a given (object, subject, action) triple; and (ii) determines the system's response (either yes or no) to every possible access request. The policy forces a yes or no decision in the presence of conflicts as well as in the absence of explicit specifications for the access.

–A set of *integrity constraints* that may impose restrictions on the content and output of the other components.[3]

Our architecture assumes that accesses are always made by a user, possibly playing a role. Access control is enforced with respect to the user's identity, if no role is active, and with respect to the role if a role is active.[4] In other words, when playing a role, the user loses his personal privileges and restrictions (that may have been granted to him specifically or to groups to which he belongs), assuming instead those of the role. From the point of view of the access control, therefore, every request is seen as coming from either a user or a role. In the following, we generically refer to the requester as subject. When a subject s requests permission to execute action a on object o, we need to check if the authorization $(o, s, +a)$ can be derived using the authorization table, history table, propagation policy, conflict resolution policy, and decision policy that are in force. If so, and if no "dynamic" constraint is violated, the access is allowed. If $(o, s, -a)$ is derivable, then the access is denied. Clearly, we must design our system in such a way that given any triple (o, s, a) *exactly one of the authorizations* $(o, s, +a)$, $(o, s, -a)$ *is derivable* from the authorization table, propagation policy and conflict resolution and decision policy in force.

In preceding sections, we have provided several examples of various propagation strategies, conflict resolution strategies, and decision strategies that the SSO might want to use. However, these represent only some of the possibilities. In reality, these strategies may be highly application dependent. Thus, a framework for authorization management must be flexible enough to allow the system security officer to express what he needs for his application. In particular, we realize the functional architecture through the following approach:

–The authorization table is viewed as a database.

–Policies are expressed by a tightly restricted class of logic programs called authorization specifications. These programs will be guaranteed to have the nice properties we describe below.

–The semantics of authorization specifications is given through the well known stable model semantics [Gelfond and Lifschitz 1988] and well-founded model semantics [van Gelder 1989] of logic programs. In fact, as we will show, the structure of authorization specifications guarantee that their stable model semantics is equivalent to their well founded semantics, thus ensuring that they have exactly one stable model (this follows from a result of Baral and Subrahmanian [1992]).

–Accesses will be allowed or denied based on the truth value of an atom associated with that access in this unique stable model. This guarantees that for any access, exactly one decision (allowed/denied) is provided by our architecture.

Accordingly, we present a logical language (in a fragment of Datalog with negation), called authorization specification language (**ASL**), that the system

[3]We will elaborate more on this in Section 4.7.
[4]We assume each user can activate at most one role at a time.

security officer can use to encode his security needs. We further show how the strategies we have described above can be expressed in this Datalog language.

4.2 The Authorization Specification Language

We assume that **DS** is an arbitrary, but fixed data system that the system security officer is specifying authorizations for. **ASL** is a logical language created from the following alphabet:

(1) *Constant Symbols*: Every member of Obj ∪ T ∪ U ∪ G ∪ R ∪ A ∪ SA. Recall that Obj is the set of objects, T the set of types, U the set of users, G the set of groups, R the set of roles, and A and SA the set of unsigned and signed actions respectively.

(2) *Variable Symbols*: There are seven sets $V_o, V_t, V_u, V_g, V_r, V_a, V_{sa}$ of variable symbols ranging over the sets Obj, T, U, G, R, A, and SA, respectively. In the remainder of the paper, the following terminology will be used. Variables in V_o and members of Obj are object terms. Variables in V_t and members of T are type terms. Variables in V_u and members of U are user terms. Variables in V_g and members of G are group terms. Variables in V_r, members of R, and constant symbol ϵ are role terms. Variables in V_a and members of A are action terms. Variables in V_{sa} and members of SA are signed action terms. We collectively refer to object, type, and role terms as authorization object terms and to user, group, and role terms as authorization subject terms.

(3) *Predicate Symbols*: **ASL** contains the following predicate symbols:

   (a) A ternary predicate symbol, cando. The first argument of cando is an authorization object term, the second is an authorization subject term, and the third is a signed action term. The predicate cando represents authorizations explicitly inserted by the SSO. They represent the accesses that the SSO wishes to allow or deny (depending on the sign associated with the action).

   (b) A ternary predicate symbol, dercando, with the same arguments as cando. The predicate dercando represents authorizations derived by the system using logical rules of inference (modus ponens plus rules for stratified negation [Apt et al. 1988]).

   (c) A 3-ary predicate symbol, do, with the same arguments as cando and dercando. It definitely represents the accesses that must be granted or denied. Intuitively, do enforces the conflict resolution and access decision policy.

   (d) A 5-ary predicate symbol, done. The first argument is an object term, the second argument is a user term, the third argument is a role term, the fourth argument is an unsigned action term, and the fifth argument is a natural number. *Intuitively,* done(o, u, r, a, t) *is true if user* u *with role* r *active has executed action* a *on object* o *at time* t.

   (e) Two 4-ary predicate symbols $\texttt{over}_{\mathsf{AO}}$ and $\texttt{over}_{\mathsf{AS}}$. $\texttt{over}_{\mathsf{AO}}$ takes as arguments two object terms, a subject term, and a signed action term. $\texttt{over}_{\mathsf{AS}}$ takes as arguments a subject term, an object term, another subject term,

and a signed action term. They are needed in the definition of some of the overriding policies.

(f) A propositional symbol `error`. If `error` can be derived through some rule, then there is an error in the specification or use of authorizations due to the satisfaction of the conditions stated in the body of the rule. Intuitively, `error` signals the violation of the integrity constraints.

The following predicate symbols, which we call `hie`-*predicates*, can be used in the specifications:

(a) A ternary predicate symbol `in` that takes as arguments two elements of $\mathsf{AO} \cup \mathsf{AS}$ and whose third argument is a ground term equal to either AOH or ASH. It captures the ordering relationships in the AOH and in the ASH hierarchies.
(b) A ternary predicate `dirin` that takes as arguments two elements such that both belong to either AO or AS and whose third argument is a ground term equal to either AOH or ASH. It captures the direct membership relationships in the AOH and in the ASH hierarchies.

In addition, a set of application-specific predicates, which we call `rel`-*predicates*, can be used in the specifications. These predicates capture the possible different relationships, existing between the elements of the data system, that may need to be taken into account by the access control system (see Section 2.2.2). Examples of such predicates, which are application specific and not fixed by the model, are:

(a) A binary predicate `owner` whose first argument is an object/type term and second argument is an authorization subject term. It associates a *unique* user or role (not group) with each object, called the *owner* of that object.
(b) Unary predicates `isuser`, `isgroup`, `isrole` whose only argument is a subject and which return true if the subject is a user, a group, or a role, respectively. They can be used for the selective application (depending on the kind of subjects) of propagation, conflict resolution, or decision policies.

If p is one of the above predicate symbols with arity n, and $t_1, \ldots, t_n$ are terms appropriate for p (as defined above), then $p(t_1, \ldots, t_n)$ is an *atom*. We use the word *literal* to denote an atom or its negation. For instance, if OT, ST and SAT are an authorization object, authorization subject, and a signed action term, respectively, then $\mathtt{cando}(OT, ST, SAT)$ and $\neg\mathtt{cando}(OT, ST, SAT)$ are examples of literals. We use the expression `over` predicate to denote either $\mathtt{over}_{\mathsf{AS}}$ or $\mathtt{over}_{\mathsf{AO}}$ indiscriminately. We use the term `rel`-literal when the predicate involved is a `rel`-predicate. We use the term, `hie`-literal, when the predicate involved is a `hie`-predicate.

Definition 4.1 (*Satisfaction of* `in` *and* `dirin` *Formulas*). Let $\mathsf{H} = (\mathsf{X}, \mathsf{Y}, \leq_{\mathsf{H}})$ be a hierarchy whose nodes $(\mathsf{X} \cup \mathsf{Y})$ are ordered by relationship $\leq_{\mathsf{H}}$.

(1) H satisfies $\mathtt{in}(x, y, \mathsf{H})$ for $x, y \in \mathsf{X} \cup \mathsf{Y}$ iff $x \leq_{\mathsf{H}} y$.

(2) H satisfies $\texttt{dirin}(x, y, \mathsf{H})$ for $x, y \in \mathsf{X} \cup \mathsf{Y}$ iff $x \leq_{\mathsf{H}} y$ and $\forall z \in \mathsf{X} \cup \mathsf{Y} : x \leq_{\mathsf{H}} z, z \leq_{\mathsf{H}} y, z \neq y \Rightarrow z = x$.
(3) If L is a nonground in or dirin, (i.e., if L contains one or more variables), then
    (a) H satisfies $(\forall x)L$ iff H satisfies every ground instance of L.
    (b) H satisfies $(\exists x)L$ iff H satisfies some ground instance of L.
(4) H satisfies a conjunction (respectively, disjunction) of in/dirin-literals iff H satisfies each (respectively, some) literal in the conjunction (respectively, disjunction).

As our language for authorization specifications is a logical language, the system security officer may specify access authorizations through rules expressed in this language. Below, we show how each aspect of the architecture of Figure 7 is encoded by a set of rules, each obeying a highly restricted syntactic form. By keeping the syntactic form tightly restricted, we are able to guarantee efficient complexity results.

4.3 History Table

The history table is recorded through a predicate called done having the arguments

$$(\texttt{Object}, \texttt{User}, \texttt{Role}, \texttt{Action}, \texttt{Time})$$

specifying that a given user playing a given role executed a given action on the specified object at a stated point in time. The instances of the done-predicate are represented as a relational table with the above schema.

Histories are useful in implementing those policies in which future accesses of users are based on the accesses each user has exercised in the past (as in the case of the Chinese Wall policy [Brewer and Nash 1989], which requires both the done predicate and the error predicate to represent it).

Note that, even in case of actions executed through a role, the executing user is recorded. Besides ensuring accountability, this allows the enforcement of separation of duty constraints at the user-level even when users are operating through roles, as will be discussed in Section 4.7.

4.4 Authorization Table

The authorization table is created by defining a view on top of done, hie-literals and rel-literals.

Definition 4.2 (*Authorization Rule*). An *authorization rule* is a rule of the form:

$$\texttt{cando}(o, s, \langle sign \rangle a) \leftarrow L_1 \& \cdots \& L_n.$$

where o, s and a are elements of AO, AS and A respectively, $n \geq 0$, $\langle sign \rangle$ is either $+$ or $-$, and $L_1, \ldots, L_n$ are done, hie-, or rel-literals.

Definition 4.3 (*Authorization View*). An *authorization view* is a finite set of authorization rules.

Definition 4.4 (*Authorization Table*). Suppose V is an authorization view. The *authorization table based on* V is the result of materializing V.

We are now ready to give a few examples of authorization views.

Example 4.1. Consider the following authorization rules:

$$\begin{aligned}
\texttt{cando}(o, \texttt{john}, +\texttt{read}) &\leftarrow \texttt{in}(o, \texttt{Letters}, \mathsf{AOH}).\\
\texttt{cando}(\texttt{file1}, \texttt{john}, +\texttt{read}) &\leftarrow .\\
\texttt{cando}(\texttt{file2}, \texttt{Citizens}, +\texttt{read}) &\leftarrow .\\
\texttt{cando}(\texttt{cobol-manual}, \texttt{Research-Staff}, +\texttt{read}) &\leftarrow .\\
\texttt{cando}(\texttt{Faculty}, \texttt{CS-Faculty}, \texttt{activate}) &\leftarrow .
\end{aligned}$$

The first rule states that `john` can read all objects of type `Letters`. The second rule states that `john` can read `file1`. The third rule states that group `Citizens` can read object `file2`. The fourth rule states that role `Research-Staff` can read `cobol-manual`. Finally, the last rule states that group `CS-Faculty` can act in role `Faculty`.

The reader may wonder: if an authorization has already been specified for a group (e.g., `Employees`), why are statements involving `in` allowed in the bodies of rules? The reason is that (as discussed in Section 3.2) there are many different approaches to *propagating* authorizations from a group to a subgroup, to a sub-subgroup, and eventually to an individual. Modeling this propagation process requires the ability to distinguish between the authorizations explicitly given to users and the authorizations they may hold as members of some group. In general, specifying an authorization for a group is different from specifying an authorization for all members of the group. To illustrate, consider the rule "$\texttt{cando}(\texttt{file1}, s, +\texttt{write}) \leftarrow \texttt{in}(s, \texttt{Employees}, \mathsf{ASH})$". This rule serves as shorthand that specifies a `cando` fact for each member of the group. Consider instead the rule "$\texttt{cando}(\texttt{file1}, \texttt{Employees}, +\texttt{write}) \leftarrow .$". This rule specifies that the group `Employees` is authorized to write `file1`. Note that this rule makes a general statement about the group as a whole, but not about specific members of the group. *Whether this authorization propagates to the members will depend on the specific policy to be applied as well as on the other authorizations specified.* Section 4.5 describes how propagation policies are implemented.

Example 4.2. Consider the following rules:

$$\begin{aligned}
\texttt{cando}(\texttt{file1}, s, +\texttt{read}) \leftarrow\ & \texttt{in}(s, \texttt{Employees}, \mathsf{ASH})\ \&\\
& \neg\texttt{in}(s, \texttt{Soft-Developers}, \mathsf{ASH}).\\
\texttt{cando}(\texttt{file2}, s, +\texttt{read}) \leftarrow\ & \texttt{in}(s, \texttt{Employees}, \mathsf{ASH})\ \&\\
& \texttt{in}(s, \texttt{Non-citizens}, \mathsf{ASH}).
\end{aligned}$$

The first rule states that all subjects belonging to `Employees` but not to `Soft-Developers` are authorized to read `file1`. The second rule states that all subjects belonging to both `Employees` and `Non-citizens` can read `file2`.

4.5 Propagation Policies

Authorization views explicitly specify what subjects can and cannot do. However, as described above, when a human user requests permission to execute an action, if no authorization is explicitly specified for the user with respect to the attempted access, there must be some way to propagate authorizations "downward." Different paths in the AOH and ASH hierarchies may propagate different authorizations to the user (and how these are resolved will be discussed in Section 4.6). For now, we specify how authorizations are propagated down specific paths.

Before introducing the rule enforcing propagation, we introduce a rule that will be useful in the definition of many propagation policies, as we will see shortly.

Definition 4.5 (*Overriding Rule*). An *overriding rule* is a rule of one of the following two forms:

$$\texttt{over}_{\mathsf{AS}}(s, o, s', \langle \text{sign} \rangle a) \ \leftarrow\ L_1 \& \ \cdots \& \ L_n.$$
$$\texttt{over}_{\mathsf{AO}}(o, o', s, \langle \textit{sign} \rangle a) \ \leftarrow\ L_1 \& \ \cdots \& \ L_n.$$

where o *and* o' are elements of AO*,* s and s' are elements of AS*,* a is an element of A*,* $\langle \textit{sign} \rangle$ is either $+$ or $-$*,* and $L_1, \ldots, L_n$ are either `cando`*,* `done`*,* `hie-`*, or* `rel-`literals.

Propagation policies can then be defined by the system security officer through *derivation rules*, defined as follows.

Definition 4.6 (*Derivation Rule*). A *derivation rule* is a rule of the form:

$$\texttt{dercando}(o, s, \langle \textit{sign} \rangle a) \ \leftarrow\ L_1 \& \ \cdots \& \ L_n.$$

where o, s and a are elements of AO, AS and A respectively, $\langle \textit{sign} \rangle$ is either $+$ or $-$, and $L_1, \ldots, L_n$ are either `cando`, `over`, `dercando`, `done`, `hie-`, or `rel-`literals. All `dercando`-literals appearing in the body of a derivation rule must be positive.

Definition 4.7 (*Derivation View*). A *derivation view* is a finite set of derivation rules.

Derivation rules may be used to express a wide variety of propagation policies, several of which are shown in Figure 8. These rules encode the policies described earlier in Section 4, thus showing that the language of derivation rules is rich enough to allow many different policies to be easily represented. Though Figure 8 shows how some well-known propagation policies may be expressed within the syntax of derivation rules, these are only a small fraction of the policies that can be expressed using derivation views. Another example of how derivation rules may be used to implement application-specific propagation policies is shown below.

Example 4.3. Consider the following derivation rules:

$$\texttt{dercando}(\texttt{file1}, s, -\texttt{write}) \ \leftarrow\ \texttt{dercando}(\texttt{file2}, s, \texttt{read}).$$

Policy	Rules
No propagation	$\texttt{dercando}(o,s,+a) \leftarrow \texttt{cando}(o,s,+a).$
	$\texttt{dercando}(o,s,-a) \leftarrow \texttt{cando}(o,s,-a).$
No overriding	$\texttt{dercando}(o,s,+a) \leftarrow \texttt{cando}(o,s',+a) \& \ \texttt{in}(s,s',\mathsf{ASH}).$
	$\texttt{dercando}(o,s,-a) \leftarrow \texttt{cando}(o,s',-a) \& \ \texttt{in}(s,s',\mathsf{ASH}).$
Most specific overrides	$\texttt{dercando}(o,s,+a) \leftarrow \texttt{cando}(o,s',+a) \& \ \neg\texttt{over}_{\mathsf{AS}}(s,o,s',+a) \& \ \texttt{in}(s,s',\mathsf{ASH}).$
	$\texttt{dercando}(o,s,-a) \leftarrow \texttt{cando}(o,s',-a) \& \ \neg\texttt{over}_{\mathsf{AS}}(s,o,s',-a) \& \ \texttt{in}(s,s',\mathsf{ASH}).$
	$\texttt{over}_{\mathsf{AS}}(s,o,s',+a) \leftarrow \texttt{cando}(o,s'',-a) \& \ \texttt{in}(s,s'',\mathsf{ASH}) \& \ \texttt{in}(s'',s',\mathsf{ASH}) \& \ s'' \neq s'.$
	$\texttt{over}_{\mathsf{AS}}(s,o,s',-a) \leftarrow \texttt{cando}(o,s'',+a) \& \ \texttt{in}(s,s'',\mathsf{ASH}) \& \ \texttt{in}(s'',s',\mathsf{ASH}) \& \ s'' \neq s'.$
Path overrides	$\texttt{dercando}(o,s,+a) \leftarrow \texttt{cando}(o,s,+a).$
	$\texttt{dercando}(o,s,-a) \leftarrow \texttt{cando}(o,s,-a).$
	$\texttt{dercando}(o,s,+a) \leftarrow \texttt{dercando}(o,s',+a) \ \& \ \neg\texttt{cando}(o,s,-a) \ \& \ \texttt{dirin}(s,s').$
	$\texttt{dercando}(o,s,-a) \leftarrow \texttt{dercando}(o,s',-a) \ \& \ \neg\texttt{cando}(o,s,+a) \& \ \texttt{dirin}(s,s').$

Fig. 8. Rules enforcing different propagation policies on ASH.

$$\texttt{dercando}(o, s, -\texttt{grade}) \leftarrow \texttt{done}(o, s, r, \texttt{write}, t) \ \& \ \texttt{in}(o, \texttt{Exams}, \mathsf{AOH}).$$
$$\texttt{dercando}(\texttt{file1}, s, -\texttt{read}) \leftarrow \texttt{dercando}(\texttt{file2}, s', \texttt{read}) \ \& \ \texttt{in}(s, g, \mathsf{ASH}) \ \& \ \texttt{in}(s', g, \mathsf{ASH}).$$

The first rule derives a denial for a subject to write `file1` if there exists an authorization (explicit or derived) for the subject to read `file2`. The second rule derives a denial for a user to grade an object of type `Exams` if the user has written that object. The third rule derives a negative authorization for a user/group s to read `file1` if there exist a user/group s' and a group g such that s and s' both belong to g and s' is authorized to read `file2`.

In general, using derivation views, a system security officer can express very general policies for propagation of authorizations down a hierarchy. However, it is entirely possible that propagation of authorizations along a hierarchy still leads to conflicts. For instance, given a derivation view, it is entirely possible that the materialization of the view includes atoms of the form

$$\texttt{dercando(o,s,+a)}, \texttt{dercando(o,s,-a)}.$$

Intuitively, this situation reflects a conflict in the authorizations associated with user s's attempt to execute action a on object o. Conflict resolution policies, which will be studied next, are used to handle such conflicts.

4.6 Conflict Resolution and Decision Policies

As we have observed earlier, the fact that the system security officer has specified a history table (implicitly by logging transactions), an authorization view, and a propagation view, still leaves open the possibility that a given attempted access leads to contradictory authorizations derived from different paths in the hierarchies being considered. The concept of a *decision view* below allows the system security officer to specify how conflicts are to be resolved.

Definition 4.8 (*Positive Decision Rule*). A *positive decision rule* is a rule of the form

$$\texttt{do}(o, s, +a) \leftarrow L_1 \& \cdots \& L_n$$

where o, s and a are elements of AO, AS, and A, respectively, and $L_1, \ldots, L_n$ are `cando`, `dercando`, `done`, `hie-` or `rel-` literals and every variable that appears in any of the L_i's also appears in the head of this rule.

It is important to note that positive decision rules allow negated literals to appear in the body of the rule – however, since recursion is not allowed, negated atoms of the form $\neg\texttt{do}(\cdots)$ cannot occur in rule bodies. Also, no negated action of the form $-a$ is allowed to appear in the head of a positive decision rule. A decision view, defined below, allows this to happen in a very restricted fashion.

Definition 4.9 (*Decision Views*). A *decision view* is a finite set of positive decision rules, together with the one additional rule:

$$\texttt{do}(o, s, -a) \leftarrow \neg\texttt{do}(o, s, +a).$$

Intuitively, the set of atoms of the form $\texttt{do}(o, s, +a)$ obtained by materializing a decision view specifies *the* set of all authorized accesses. The system security officer can specify *any* set of positive decision rules that he wishes–the system will not stop him (unless he violates a syntactic condition). However, once he finishes specifying his positive decision rules, the system will automatically add the one negative rule shown above. This special rule says that the only authorizations granted are those that are explicitly derived after conflict resolution. The additional rule therefore guarantees completeness of the specifications. Its form also guarantees no conflict. In other words, for every possible request $(o, s, +a)$ either $\texttt{do}(o, s, +a)$ or $\texttt{do}(o, s, -a)$ will hold, and the request will therefore be granted or denied accordingly.

The following example shows one decision view that a system security officer might specify.

Example 4.4. Consider the following rules:

$$\texttt{do}(\texttt{file1}, s, +a) \leftarrow \texttt{dercando}(\texttt{file1}, s, +a).$$

$$\texttt{do}(\texttt{file2}, s, +a) \leftarrow \neg\texttt{dercando}(\texttt{file2}, s, -a).$$

$$\texttt{do}(o, s, +\texttt{read}) \leftarrow \neg\texttt{dercando}(o, s, +\texttt{read}) \& \neg\texttt{dercando}(o, s, -\texttt{read}) \& \texttt{in}(o, \texttt{Pblc-docs}, \mathsf{AOH}).$$

The first rule states that a subject (user or role) can exercise an access on object `file1` if he has a positive authorization for it (i.e., the closed policy is enforced on `file1`). The second rule states that a subject can exercise an access on `file2` if he does not have a negative authorization for it (i.e., the open policy is enforced on `file2`). The last rule states that if no authorization has been derived for a subject on an object of type `Pblc-docs`, the subject can read the object.

The difference between decision rules, and `cando` and `dercando` rules is that these latter rules specify authorizations (either explicitly or derived) provided

CONFLICT	DECISION	RULES
No conflict	open	error ← dercando($o, s, +a$)& dercando($o, s, -a$). do($o, s, +a$) ← ¬dercando($o, s, -a$).
No conflict	closed	error ← dercando($o, s, +a$)& dercando($o, s, -a$). do($o, s, +a$) ← dercando($o, s, +a$)& ¬dercando($o, s, -a$).
Denials take p.	open	do($o, s, +a$) ← ¬dercando($o, s, -a$).
Denials take p.	closed	do($o, s, +a$) ← dercando($o, s, +a$)& ¬dercando($o, s, -a$).
Permissions take p.	open	do($o, s, +a$) ← dercando($o, s, +a$). do($o, s, +a$) ← ¬dercando($o, s, -a$).
Permissions take p.	closed	do($o, s, +a$) ← dercando($o, s, +a$).
Nothing takes p.	open	do($o, s, +a$) ← ¬dercando($o, s, -a$).
Nothing takes p.	closed	do($o, s, +a$) ← dercando($o, s, +a$)& ¬dercando($o, s, -a$).
Additional closure rule		do($o, s, -a$) ← ¬do($o, s, +a$).

Fig. 9. Possible rules enforcing different conflict resolution and decision policies.

by the system security officer. In contrast, decision (do) rules specify how unresolved conflicts are to be resolved, and what the system must do in response to a specific request. Figure 9 provides a list of policies for conflict resolution and decision that can easily be expressed through the mechanism of decision rules.

4.7 Integrity Rules

Authorization, derivation, and decision rules defined in the preceding subsections are all we need to specify authorizations and access control decisions. However, there is great potential for errors to arise in authorization specifications – such errors could be in hierarchies (e.g., John may be erroneously listed as belonging to both group `Citizens` and group `Non-citizens`) or in the specification of `cando` (e.g., we may have a statement of the form `cando(o,s,+a)` and `cando(o,s,-a)`). Even in their current form, authorization specifications guarantee that such errors will not cause both do(o, s, $+a$) and do(o, s, $-a$) to be derivable. However, to identify and eliminate such errors, we may extend authorization specifications through an additional type of rule, called the *integrity* rule, by which the SSO can define constraints that must hold on the authorization specifications or on the actual access execution. Integrity rules are formally defined as follows:

Definition 4.10 (*Integrity Rule*). An *integrity rule* is a rule of the form

$$\texttt{error} \leftarrow L_1\& \ \ldots \ \& L_n.$$

where $L_1, \ldots, L_n$ are `cando`, `dercando`, `done`, `do`, `hie-` or `rel-` literals.

An integrity rule derives an error every time the conditions in the right hand side of the rule are satisfied. Integrity rules provide a powerful way to specify restrictions on authorizations specification and derivation, on access control decision, as well as on the actual execution of actions by subjects. Restrictions may be general or specific to an application. General rules control inconsistencies such as "not both a positive and a negative authorization should be specified or derived for the same access" (in the case where the no conflict policy is enforced). Application dependent rules control inconsistencies specified by an application, such as "a subject cannot be authorized to read both `fileA` and `fileB`." The following example illustrates some restrictions that can be easily captured via error rules.

Example 4.5. Consider the following integrity rules:

$$\texttt{error} \leftarrow \texttt{in}(s, \texttt{Citizens}, \mathsf{ASH})\ \&\ \texttt{in}(s, \texttt{Non-citizens}, \mathsf{ASH}).$$

$$\texttt{error} \leftarrow \texttt{cando}(o, s, +a)\ \&\ \texttt{cando}(o, s, -a).$$

$$\texttt{error} \leftarrow \texttt{do}(o, s, +\texttt{write})\ \&\ \texttt{do}(o, s, +\texttt{evaluate})\ \&\ \texttt{in}(o, \texttt{Tech-reports}, \mathsf{AOH}).$$

$$\texttt{error} \leftarrow \texttt{done}(o, u, r, \texttt{submit}, t)\ \&\ \texttt{done}(o, u, r', \texttt{approve}, t')\ \&\ \texttt{done}(o, u, r'', \texttt{pay}, t'')\ \&\ \texttt{in}(o, \texttt{Order}, \mathsf{AOH}).$$

$$\texttt{error} \leftarrow \texttt{done}(o, u, r, \texttt{read}, t)\ \&\ \texttt{done}(o', u, r', \texttt{read}, t')\ \&\ \texttt{in}(o, \texttt{Budget-A}, \mathsf{AOH})\ \&\ \texttt{in}(o', \texttt{Budget-B}, \mathsf{AOH}).$$

The first rule states that a user/group cannot belong to both groups `Citizens` and `Non-Citizens`. The rule avoids clearly inconsistent membership specifications. The second rule states that the SSO cannot specify both a positive and a negative authorization for a given access (o, s, a). The third rule returns an error if a subject is authorized (notice by the decision policy!) to both `write` and `evaluate` an object of type `Tech-reports` (e.g., the author of a paper should not referee that paper). The fourth rule states that a user cannot execute all three operations, `submit`, `approve`, and `pay` on an object of type `Order`. Finally, the last rule states that a user cannot read both an object of type `Budget-A` and an object of type `Budget-B`.

The example shows how error rules can conveniently express many restrictions that may need to be enforced. The first rule is an example of a restriction that can be specified on the hierarchy topology and, in particular, on the group configuration. The second and third rules are examples of restrictions on the authorization specification, and in particular on the authorizations explicitly granted (second rule) and on those actually holding (third rule). Similar rules can be specified on derived authorizations. Note that the third rule is an example of the application of the *static* separation of duty principle, which supports the need to restrict the privileges for which a given subject can hold authorization. The latter two rules specify constraints on the actual execution of a sequence of actions by the same user, regardless of whether the user can be authorized for the action either personally, or as a member of a group, or because of the role he is playing. These rules are examples of *dynamic* separation of duty, restricting the accesses that subjects can actually execute, rather than those for which they can hold authorizations. Note the difference between static separation of duties and dynamic separation of duties. When considering separation of duties, if, for instance, the same user cannot be authorized for two accesses, the SSO may use his/her discretion to statically authorize one of the two accesses. On the other hand, with the dynamic separation of duty approach, the SSO can say that the system can authorize the user to make one access, but not both: When the user exercises one of the accesses, the error rule will automatically rule out the possibility for the user to exercise the other. For instance, the dynamic separation of duty constraint expressed in the fourth rule can be applied when several users (with interchanging tasks) can execute the mentioned operations but at least two users must participate in the completion of the ordering process. Notice also that the last rule enforces a particular type

of dynamic separation of duty, called the Chinese Wall Policy [Brewer and Nash 1989].

These are only some examples of rules that can be enforced; a variety of others can be imagined. For instance, all rules given above constrain the authorizations (or the accesses) of a user. Rules can also be specified to restrict authorizations (or accesses) for different users, objects, and so on. For instance, the third rule in the example could be modified to refer to two different subjects (s and s') in the same group (in(s, G, ASH) and in(s', G, ASH)), thus forbidding users to evaluate papers of authors that belong to the same group.

Integrity rules constrain the content of the tables used by the authorization framework. Each operation on the system (including access requests) causes changes in some table. Every time a table is updated, integrity rules are evaluated and, if the change implies an error, the corresponding operation should be denied. With respect to access requests and the architecture in Figure 7, integrity rules are evaluated after the access decision is taken. They can block the execution of the access even if the access is allowed by the specifications. This treatment of error rules allows us to easily support possible dynamic constraints on accesses without unnecessarily complicating the authorization derivation and decision policies.

4.8 Authorization Specifications

An authorization specification **AS** consists of the following five components:

–A history table;

–An authorization view/table;

–hie-, rel-, and overriding predicates;

–A derivation view;

–A decision view;

–A set of integrity rules.

In our architecture, the system security security officer must create an authorization specification **AS**. The following results tell us that the syntax of authorization specifications is rich enough to satisfy a variety of important expressiveness criteria, yet it is sufficiently restricted to be polynomial time computable. The reader is cautioned that proofs of the theorems presented below assume a knowledge of standard results and terminology in logic programming.

THEOREM 1. *Every authorization specification* **AS** *is a locally stratified* (*not stratified*) *logic program. As a consequence:*

(1) (*The logic program version of*) **AS** *has a unique stable model;*

(2) *The stable model and well founded model semantics of* (*the logic program version of*) **AS** *coincide;*

(3) *The unique stable model can be computed in quadratic time data complexity.*

PROOF. To see that **AS** is a locally stratified (not stratified) logic program, we start by recapitulating the syntax of the components of **AS** (Table I).

Table I. Syntax of the components of **AS**

Predicate	Rules defining predicate
hie-predicates rel-predicates done	base relations. base relations. base relation.
cando	body may contain done, hie- and rel-literals.
over	body may contain cando, done, hie- and rel-literals.
dercando	body may contain cando, over, dercando, done, hie-, and rel- literals. Occurrences of dercando literals must be positive.
do do	in the case when head is of the form do(_, _, +a) body may contain cando, dercando, done, hie- and rel- literals. in the case when head is of the form do(o, s, −a) body contains just one literal ¬do(o, s, +a).
error	body may contain do, dercando, cando, done, hie- and rel-literals.

It is easy to see that in general, authorization specifications are not stratified [Apt et al. 1988], but are always locally stratified. A local stratification may be obtained through the following level mapping [Apt et al. 1988; Przymusinski 1988] ℓ that assigns: level 0 to all atoms involving hie-predicates, rel-predicates, and done; level 1 to all atoms of the form cando(_, _, _); level 2 to all atoms of the form over(_, _, _, _); level 3 to all atoms of the form dercando(_, _, _); level 4 to all atoms of the form do(_, _, $+a$); level 5 to all atoms of the form do(_, _, $-a$); and level 6 to the atom error. ℓ is a witness to the local stratifiability of **AS**. We have thus shown at this stage that **AS** is a locally stratified logic program. Now,

(1) follows immediately from a result of Gelfond and Lifschitz [1988] showing that all locally stratified logic programs have a unique stable model.
(2) follows immediately from a result of Baral and Subrahmanian [1992] showing that for locally stratified logic programs, well-founded semantics coincides with the unique stable model of the program.
(3) follows immediately from the result of Van Gelder et al. [1989] showing that well-founded semantics can be computed in quadratic time (data complexity). □

The following result tells us that authorization specifications are both decisive (i.e. given any triple (o, s, a), either do$(o, s, +a)$ or do$(o, s, -a)$ is true in the unique stable model of **AS**) and conflict-free (i.e., there is no triple (o, s, a) such that both do$(o, s, +a)$ and do$(o, s, -a)$ are true in the unique stable model of **AS**).

THEOREM 2. *Suppose* **AS** *is an authorization specification, and a user s attempts to execute action a on an object o. Then: exactly one of the two atoms* do(o,s,+a)*,* do(o,s,−a) *is true in the unique stable model of* **AS**.

PROOF [Consistency]. Suppose the unique stable model M of **AS** contains both do(o,s,−a) and do(o,s,+a) for some triple (o, s, a). As all stable models of **AS** are supported models [Marek and Subrahmanian 1992], it follows from

the structure of **AS** that there exists a ground instance of a rule in **AS** having the form

$$\mathtt{do}(o,s,-a) \leftarrow \neg\mathtt{do}(o,s,+a)$$

whose body is true in M. This is impossible as $\mathtt{do}(o,s,+a) \in M$ by our assumption that M contains both $\mathtt{do(o,s,-a)}$ and $\mathtt{do(o,s,+a)}$, thus leading to a contradiction.

[Decisiveness] Suppose M does not contain the ground atom $\mathtt{do(o,s,+a)}$. Then, the Gelfond–Lifschitz transform[5] of **AS** contains the ground rule

$$\mathtt{do}(o,s,-a) \leftarrow .$$

As M is stable, we know that $M = T_{GL(\mathbf{AS},M)} \uparrow \omega$, where $GL(\mathbf{AS}, M)$ denotes the Gelfond Lifschitz transform of **AS** with respect to M and T denotes the standard immediate consequence operator [Lloyd 1987]. It is immediate that $\mathtt{do}(o,s,-a) \in T_{GL(\mathbf{AS},M)} \uparrow \omega = M$, showing that when M does not contain the ground atom $\mathtt{do(o,s,+a)}$, it must contain $\mathtt{do}(o,s,-a)$. □

Before concluding this section, we provide a brief comparison with the work of Woo and Lam [1993] who propose the use of default logic [Reiter 1980] to express authorization and control rules. Recall that in our case, an authorization specification is a syntactically restricted logic program. Consider the following translation $\mathcal{T}$ that converts an authorization specification P into a default theory[6] $\mathcal{T}(P) = (\emptyset, D)$ as follows, where:

$$D = \left\{ \frac{b_1, \ldots, b_n : \neg c_1, \ldots, \neg c_m}{a} \;\middle|\; \begin{array}{l} a \leftarrow b_1 \,\&\, \cdots \,\&\, b_n \,\&\, \neg c_1 \,\&\, \cdots \,\&\, \neg c_m \\ \text{is a ground instance of a rule/fact in } P \end{array} \right\}.$$

The following result establishes the fact that all authorization specifications may be equivalently expressed in default logic.

THEOREM 3. *Suppose P is an authorization specification. Then:*

(1) *$\mathcal{T}(P)$ has a unique extension, which we denote by $E_{\mathcal{T}(P)}$.*
(2) *A ground atom A is true in the unique stable model of P* iff $A \in E_{\mathcal{T}(P)}$.

[5]The Gelfond-Lifschitz transform of **AS** with respect to interpretation M, denoted as $GL(\mathbf{AS}, M)$, is the set of rules obtained as follows: (1) Consider the ground version $\mathbf{AS}^G$ of **AS**. (2) Remove from $\mathbf{AS}^G$ all the rules whose body contains a negative literal $\neg L$ such that $L \in M$. (3) Remove from all the remaining rules all the negative literals. Given a ground program P, the standard immediate consequence operator is a mapping from interpretations to interpretations defined as $T_P(I) = \{A \mid (A \leftarrow L_1 \& \ \ldots \ \& L_k) \in P$, and $I \models L_i, i = 1, \ldots, k\}$.

[6]A *default theory* is a pair (D, W), where W is a set of first order formulas and D is a set of defaults of the form $\frac{A:B_1,\ldots,B_n}{C}$, where A, B_i, C are classical formulas, $i = 1, \ldots, n$. A set of formulae E is an extension of (D, W) iff E is a fixed point of the operator Γ, defined as follows: given a set S of formulas, $\Gamma(S)$ is the smallest set such that (i) $W \subseteq \Gamma(S)$, (ii) $Th(\Gamma(S)) = S$ (i.e., S is deductively closed), and (iii) if $\frac{A:B_1,\ldots,B_n}{C}$ belongs to D, $A \in \Gamma(S)$, $\neg B_i \notin S$ $(i = 1, \ldots, n)$, then $C \in \Gamma(S)$.

PROOF. Marek and Subrahmanian [1992] show that M is a stable model of P iff $\mathcal{T}(P)$ has all consequence, $Cn(M)$, as an extension. As P has only one stable model, M, $Cn(M)$ is an extension of $\mathcal{T}(P)$ which satisfies condition (1). Condition (2) follows immediately because the only ground atoms true in $Cn(M)$ are those in M. □

The above theorem has the following impact. It shows that all authorization specifications may be expressed as default logic theories. In contrast, all default logic theories cannot be expressed as authorization specifications. To see why, take the default logic theory $\Delta = (\emptyset, \{\frac{b:c}{c}\})$. It is easy to see that there is no logic program P such that $\mathcal{T}(P) = \Delta$. This is beneficial to us, because evaluating authorization requests against an authorization problem will be shown to be polynomial. In contrast, if one uses a general default logic solver, it will be intractable at best, and in some cases, even undecidable [Gottlob 1992].

5. MATERIALIZING AND MAINTAINING DERIVATION AND DECISION VIEWS

The results of the preceding section guarantee that given any authorization specification **AS**, we assess a request to execute a particular privilege on a data object by checking if it is true in the unique stable model of **AS**. If so, the request is authorized, otherwise, it is denied.

However, when implementing an algorithm to support this kind of evaluation, we need to consider the following facts:

–The request should be either authorized or denied very fast.

–Changes to the specifications are far less frequent than access requests.[7]

Since access requests happen all the time, we would like a security architecture to optimize processing of these requests. For this purpose, we propose a *materialized view architecture*. At any given point t in time, we materialize the decision and derivation views. When a subject asks to execute an operation, we merely check the decision view to see if he is authorized to do so. This test is very simple indeed. If the request is authorized *and* the recording of the access in the history table would not cause any error to be derived, the access is granted (it is denied, leaving the history table invaried otherwise). Consequently the history table is modified and the update propagated to the materialized decision view and derivation view.

In this section, we first specify how to materialize the decision/derivation views of **AS**, and then show how to propagate updates to the history table.

[7]One may object here that although this is true in general (where administrative operations are inserting new rules or facts), it does not hold in our framework. The reason for this is that since our **AS** keeps a history of the accesses and rules can put conditions on such a history, every access resulting in the insertion of a new done predicate in **AS** results in fact in a change to the specification. As per the discussion forthcoming in Section 5.1, most such changes will be ineffective: very few rules will use done predicates (only those enforcing history-based controls); and after a done fact δ has been registered, insertion of subsequent done facts differing from δ only for the time component affects only those (history-based) rules with mathematical conditions on the time component.

Level	Stratum	Predicate	Rules defining predicate
0	$\mathbf{AS}_0$	hie-predicates rel-predicates done	base relations. base relations. base relation.
1	$\mathbf{AS}_1$	cando	body may contain done, hie- and rel-literals.
2	$\mathbf{AS}_2$	dercando	body may contain cando, dercando, done, hie-, and rel- literals. Occurrences of dercando literals must be positive.
3	$\mathbf{AS}_3$	do	in the case when head is of the form do(_, _, $+a$) body may contain cando, dercando, done, hie- and rel- literals.
4	$\mathbf{AS}_4$	do	in the case when head is of the form do($o, s, -a$) body contains just one literal $\neg$do($o, s, +a$).
5	$\mathbf{AS}_5$	error	body may contain do, cando, dercando, done, hie-, and rel- literals.

Fig. 10. Authorization specification strata.

5.1 Materialization Structure

Recall that an authorization specification **AS** is a locally stratified logic program, whose strata are defined as shown in Figure 10.[8] Even more, the program is "almost stratified". The only reason for nonstratification is the negative dependency between different instances of predicate do in the additional completeness rule "do($o, s, -a$) $\leftarrow \neg$do($o, s, +a$)" automatically added by the system to guarantee completeness of the specifications (see Section 4.6). However, there is no need to materialize negative do facts; the truth value of any fact do($o, s, -a$) is exactly the truth value of the negative literal $\neg$do($o, s, +a$), evaluated against the model of the lower stratum. Consequently, the materialization process ignores completeness rules, and operates on a stratified program.

To be able to incrementally update the computed materialized model upon changes to the specifications, instead of simply materializing the model of **AS**, we maintain a materialization structure that associates with each fact of the model the indexes of the rules that directly support its truth.

Definition 5.1 (*Materialization Structure*). The *Materialization Structure* for an authorization specification **AS** is a set of pairs (A, S), where A is a ground atom in the authorization specification language and S is a set of (indices of) rules whose head unifies with A.

Given a rule r, we use *head*(r) and *body* (r) to denote the head and body, respectively, of rule r.

Definition 5.2 (*Correctness of Materialization Structures*). Let **AS** be an authorization specification and let $\mathcal{MS}$ be a materialization structure. We say that $\mathcal{MS}$ *correctly models* **AS** iff for any pair $(A, S) \in \mathcal{MS}$, the following conditions hold:

[8]For the sake of simplicity, we ignore over predicates, which are specific to the path overriding policy only, to focus the attention on the most important and general predicates of the language. This is not a limitation, since over predicates do not introduce negative recursion in the program, thus they cannot cause nonstratification.

(1) $A \in \mathcal{M}(\mathbf{AS})$, that is, A belongs to the model of the authorization specification;

(2) for each $A \in \mathcal{M}(\mathbf{AS})$, there is at least one pair $(A, S) \in \mathcal{MS}$;

(3) for all rules r such that θ is the most general unifier of $head(r)$ and $A, r \in S$ iff $body(r)\theta$'s existential closure is true in $\mathcal{M}(\mathbf{AS})$.

According to the definitions above, the materialization structure that correctly models an authorization specification **AS** contains a pair for each atom A that is true in the (unique stable model) of **AS**, where element S in the pair contains all and only the indices of the rules that directly support the truth of A. As an example, the pair (`do(mail, alice, +read)`,$\{r_1, r_2\}$) in $\mathcal{MS}$ states that `do(mail, alice, +read)` holds in the model of **AS**, and that its truth follows from the satisfaction of the existential closure of the body of r_1 as well as from the satisfaction of the existential closure of the body of r_2. The set S associated with the atom plays a role in the insertion/deletion of the atom in/from the materialization. Intuitively, an atom will be deleted from the materialization only when its support becomes empty.

Changes to the materialization structure are realized through operators $\oplus$ and $\ominus$. Operator $\oplus$ enforces addition of a pair (A, S) to a materialization structure. Operator $\ominus$ enforces deletion of a pair (A, S) from a materialization structure. They are defined as follows:

$$\mathcal{MS}(\mathbf{AS}) \oplus (A, S) = \begin{cases} \mathcal{MS}(\mathbf{AS}) \cup \{(A, S)\} & \text{if } \nexists (A, S') \in \mathcal{MS}(\mathbf{AS}) \\ \mathcal{MS}(\mathbf{AS}) \setminus \{(A, S')\} \cup \{(A, S' \cup S)\} & \text{if } \exists (A, S') \in \mathcal{MS}(\mathbf{AS}) \end{cases}$$

$$\mathcal{MS}(\mathbf{AS}) \ominus (A, S) = \begin{cases} \mathcal{MS}(\mathbf{AS}) & \text{if } \nexists (A, S') \in \mathcal{MS}(\mathbf{AS}) \\ & \quad \text{such that } S \cap S' \neq \emptyset \\ \mathcal{MS}(\mathbf{AS}) \setminus \{(A, S')\} & \text{if } \exists (A, S') \in \mathcal{MS}(\mathbf{AS}) \\ & \quad \text{such that } S' \subseteq S \\ \mathcal{MS}(\mathbf{AS}) \setminus \{(A, S')\} \cup \{(A, S' \setminus S)\} & \text{if } \exists (A, S') \in \mathcal{MS}(\mathbf{AS}) \\ & \quad \text{such that } S \cap S' \neq \emptyset, \\ & \quad S' \not\subseteq S \end{cases}$$

Given a materialization structure $\mathcal{MS}$ of an authorization specification **AS**, the model $\mathcal{M}$ of **AS** is then the projection over the first element of the pairs, written $\mathcal{M} = \Pi_1(\mathcal{MS})$. In the following, we often refer to a materialization structure simply as materialization. Also, we denote with $\mathcal{MS}_i$ and $\mathcal{M}_i$ the materialization structure and the model, respectively, at stratum $\mathbf{AS}_i$.

5.2 Materialized Views

The computation of the unique stable model of an authorization specification **AS** is an iterative process that at each step i computes the least model of $\mathbf{AS}_i \cup \mathcal{M}(\mathbf{AS}_{i-1})$, where $\mathcal{M}(\mathbf{AS}_{i-1})$ is the least model of stratum

$\mathbf{AS}_{i-1}$. We now describe the different steps of this materialization computation process.

Step (0): $\mathcal{M}_0$, that is, the model of the lowest stratum, containing only positive facts that are either hie- or rel- or done predicates is given by the union of the extensions of these base relations. These facts are stored in the materialization structure $\mathcal{MS}_0 = \{(A, _)\} \mid A$ is a hie- or a rel- or a done fact$\}$, where the underscore in place of the supporting set indicates that A is a fact in the program.

Step (1): The authorization view (cando rules) is only defined in terms of *level-0 predicates*, there is no recursion in its definition. Hence, its materialization can be logically defined as follows:

$$\mathcal{MS}_1 = \{as_1\} \oplus \cdots \oplus \{as_n\},$$

where $\{as_1, \ldots, as_n\} = \{(\mathtt{cando}(o, s, \langle sign \rangle a), \{r\}) \mid$ there exist a rule r in $\mathbf{AS}$ and a grounding substitution θ such that $head(r)\theta = \mathtt{cando}(o, s, \langle sign \rangle a)$ and the join of all relations in $body(r)\theta$ is not empty$\}$.

Set $\{as_1, \ldots, as_n\}$ can be directly computed using the relational algebra, as follows: Suppose $body(r) = p_1[\vec{x}_1, \vec{y}_1] \,\&\cdots\&\, p_m[\vec{x}_m, \vec{y}_m] \,\&\, \neg p_{m+1}[\vec{x}_{m+1}, \vec{y}_{m+1}]\& \cdots \,\&\, \neg p_{m+k}[\vec{x}_{m+k}, \vec{y}_{m+k}]$ where $p_i[\vec{x}_i, \vec{y}_i]$ represents an atom of the form $p_i(\vec{t}_i)$ where the components of $\vec{t}_i$ are terms containing variables $\vec{x}_i$ that are ground instantiated by θ and other variables, $\vec{y}_i$ that are not. Then the conjunctive query represented by $body(r)\theta$ in the evaluation of $\mathcal{MS}_1$ can be expressed as the SQL query:

SELECT $\vec{y}_1, \ldots, \vec{y}_{m+k}$
FROM $p_1, \ldots, p_{m+k}$
WHERE $\vec{x}_1 = \theta(\vec{x}_1)$ **AND** $\cdots$ **AND** $\vec{x}_{m+k} = \theta(\vec{x}_{m+k})$ **AND**
⟨ join conditions defined by variables shared by atoms ⟩

Step (2): Unlike the previous step, stratum $\mathbf{AS}_2$ (i.e., the derivation view) cannot be materialized by a simple relational algebra expression. The reason for this is the possibility of (positive) recursion in the rules. We materialize $\mathbf{AS}_2$ using a differential fixpoint evaluation procedure defined as follows:

(a) For each rule r in $\mathbf{AS}_2$, split the body of r into two sets. The first set, denoted D_r, contains all dercando literals. The second set, denoted N_r, contains all the non-dercando literals. Evaluate the conjunctive query associated with the non-dercando literals against $\Pi_1(\mathcal{MS}_0 \cup \mathcal{MS}_1)$, the materialized model of the first two strata. Store the result as a materialized view V_r. Rewrite r as the rule r_{rew}:

$$head(r) \leftarrow V_r \,\&\, \bigwedge_{A \in D_r} A.$$

(b) Let $tr(\mathbf{AS}_2)$ be the set of all rules $\{r_{rew} \mid r \in \mathbf{AS}_2\}$. $tr(\mathbf{AS}_2)$ and $\mathbf{AS}_2$ are logically equivalent (see Lemma 5.1). We then compute the materialization with reference to $tr(\mathbf{AS}_2)$ instead of $\mathbf{AS}_2$. This has the advantage that even if recursive rules fire several times on different instances of the recursive predicates, we do not need to reevaluate the nonrecursive part of the bodies.

(c) Let $\mathcal{MS}$ be any materialization structure. Define the *program transformation* $\Phi_{\mathbf{AS}_2}$ as follows:

$$\Phi_{\mathbf{AS}_2}(\mathcal{MS}) = \{(\texttt{dercando}(o, s, \langle sign\rangle a), \{r\}) \mid \text{there exist a rule } r_{rew} \text{ in } \underline{tr(\mathbf{AS}_2)} \text{and a grounding substitution } \theta \text{ such that } head(r_{rew})\theta = \texttt{dercando}(o, s, \langle sign\rangle a) \text{ and } V_r \cup \Pi_1(\mathcal{MS}) \text{ satisfies the existential closure of } body(r_{rew})\theta\}.$$

Notice that, in the program transformation (and hence in the materialization structure), we keep track of the rules in the original authorization specification (r), instead of their rewritten version (r_{rew}). The reason for this is that the equivalence between a rule and its rewritten version is not guaranteed to hold in the presence of updates to lower strata. Thus, in principle, we could have many rewritten versions of each given rule, one version for each materialization of the conjunction of nonrecursive literals in the body. Our materialization structure is intended to associate each derived fact with the rules that support its truth, without discriminating between possible different instantiations, due to different rewritings, of the rules.

The language is finite, the set of all materialization structures is a complete lattice with respect to subset inclusion, and the operator $\Phi_{\mathbf{AS}_2}$ is monotonic and continuous. Every monotonic operator has a least fixpoint, which is also its least prefixpoint [Tarski 1955]. Thus, $\Phi_{\mathbf{AS}_2}$ has a least fixpoint denoted $\mathsf{lfp}(\Phi_{\mathbf{AS}_2}(\mathcal{MS}))$.

(d) Set $\mathcal{MS}_2 = \oplus\mathsf{lfp}(\Phi_{\mathbf{AS}_2}(\emptyset))$.
Notice that although $\mathcal{MS}_0$ and $\mathcal{MS}_1$ do not appear in the formula, they do play a role in the materialization of $\mathbf{AS}_2$; they have been taken into consideration during the materialization of the nonrecursive views V_r.

Step (3): The materialization process for stratum $\mathbf{AS}_3$ (i.e., the decision view) is analogous to the materialization process for $\mathbf{AS}_1$ (step (1), above). The only difference is that the evaluation of do-predicates performs the algebra query on $\Pi_1(\mathcal{MS}_0 \cup \mathcal{MS}_1 \cup \mathcal{MS}_2)$.

Step (4): As already discussed, we ignore stratum $\mathbf{AS}_4$ in the materialization. There is no need to materialize $\mathbf{AS}_4$ since the truth value of any fact $\texttt{do}(o,s,-a)$ is exactly the truth value of the negative literal $\neg\texttt{do}(o,s,+a)$ evaluated against the model $\mathcal{M}_3$.

Step (5): Integrity rules are completely defined in terms of literals of the above strata. Actually, as we have already discussed, any literal $\texttt{do}(o,s,-a)$ is equivalent to the negative literal $\neg\texttt{do}(o,s,+a)$, that can be evaluated against the model $\mathcal{M}_3$. The materialization of stratum $\mathbf{AS}_5$ is thus analogous to the materialization process for $\mathbf{AS}_1$. Specifically, the materialization process first rewrites the integrity rules, replacing any occurrence of $\texttt{do}(o,s,-a)$ with the negation of $\texttt{do}(o,s,+a)$, and then performs the algebra query on $\Pi_1(\mathcal{MS}_0 \cup \mathcal{MS}_1 \cup \mathcal{MS}_2 \cup \mathcal{MS}_3)$.

Notice that the cardinality of $\mathcal{MS}_5$ is at most one, since `error` is the unique predicate symbol defined in this stratum, with 0-arity. The reason why we

choose to have it materialized is to have the list of integrity rules supporting it, whenever error belongs to the model. This list identifies the set of integrity rules that are violated by the current authorization specification.

LEMMA 5.1. *Let r be any rule in* $\mathbf{AS}_2$ *and let* r_{rew} *be its rewritten version*

$$head(r) \leftarrow V_r \ \& \bigwedge_{A \in D_r} A,$$

where V_r *is the materialized view that stores the result of the conjunctive query associated with the non-*dercando *literals in the body of r, evaluated against the materialization of the first two strata, and* D_r *is the set of* dercando *literals in the body of r. Then, rules r and* r_{rew} *are logically equivalent.*

PROOF. Let r be a dercando rule in $\mathbf{AS}_2$. Let $D_r = \texttt{dercando}_1(tuple_1) \ \& \cdots \& \ \texttt{dercando}_k(tuple_k)$ be the conjunction of the dercando literals in $body(r)$, and $N_r = L_1 \& \cdots \& L_n$ be the conjunction of non-dercando literals in $body(r)$. For any substitution θ, $body(r)\theta$ is true in $\mathcal{M}_2$ iff $N_r\theta$ & $D_r\theta$ is true in $\mathcal{M}_2$. The truth of $N_r\theta$ in $\mathcal{M}_2$ is exactly its truth in $\mathcal{M}_1$, since the program is stratified, and $L_1, \ldots, L_n$ are all defined in the lower strata $\mathbf{AS}_0$ and $\mathbf{AS}_1$. We can thus replace $N_r\theta$ with its materialization V_r calculated against model $\mathcal{M}_1$. Then, the truth of the body of the original rule holds iff V_r & $D_r\theta$ is true in $\mathcal{M}_2$, which proves the equivalence. □

The following theorem states that the above procedure is sound and complete.

THEOREM 4. *Let* $\mathbf{AS} = \cup_{i=0,\ldots,4}\mathbf{AS}_i$ *be an authorization specification, and let* $\mathcal{MS}_i$ *be the materialization structure for stratum* $\mathbf{AS}_i$. *Then,*

$$\bigcup_{i=0,\ldots,3} \mathcal{MS}_i \textit{ correctly models } \bigcup_{i=0,\ldots,3} \mathbf{AS}_i.$$

PROOF. We prove that, $\cup_{i=0,\ldots,j}(\mathcal{MS}_i)$ correctly models $\cup_{i=0,\ldots j}(\mathbf{AS}_i)$, for any $j = 0, \ldots, 3$.

j=0) Trivial.

j=1) By construction, a pair $(\texttt{cando}(tuple), \{\ldots, r, \ldots\})$ belongs to $\mathcal{MS}_1$ iff

(a) the rule r belongs to $\mathbf{AS}_1$,
(b) $\texttt{cando}(tuple) = head(r)\theta$, for some grounding substitution θ, and
(c) The join of all the relations in $body(r)\theta$, computed against $\mathcal{M}_0$, is not empty.

These three conditions hold iff cando(*tuple*) belongs to $\mathcal{M}_1$ and its truth is supported by rule r. Hence, $\mathcal{MS}_1$ correctly models $\mathbf{AS}_1$.

j=2) From Lemma 5.1, for every rule r and grounding substitution θ, r supports the truth of $head(r)\theta$ in $\mathcal{M}_1$ iff r_{rew} does. Thus, we can equivalently consider $\mathbf{AS}_2$, or $\mathbf{AS}_2^{rew} = \{r_{rew} \mid r \in \mathbf{AS}_2\}$.

We prove that a pair $(\texttt{dercando}(tuple), \{\ldots, r, \ldots\})$ belongs to $\mathcal{MS}_2$ iff dercando(*tuple*) belongs to $\mathcal{M}_2$, and its truth is supported by rule r, that is,

$\texttt{dercando}(tuple) = head(r)\theta$, for some grounding substitution θ, and the existential closure of $body(r)\theta$ is satisfied in $\mathcal{M}_2$. To do so, we show a correspondence between the program transform T_{AS_2}, whose least fixed point is the model $\mathcal{M}_2$, and the program transformation $\Phi_{\mathbf{AS}_2}$, whose least fixed point is the materialization structure $\mathcal{MS}_2$.

By construction, $(\texttt{dercando}(tuple), \{\ldots, r, \ldots\})$ belongs to $\mathcal{MS}_2$ iff there exists an n such that $(\texttt{dercando}(tuple), \{\ldots, r, \ldots\}) \in \Phi^n_{\mathbf{AS}_2} \setminus \Phi^{n-1}_{\mathbf{AS}_2}$. Intuitively, n is the step, in the computation of the materialization structure, that adds $(\texttt{dercando}(tuple), \{r\})$ to the current partial materialization $\Phi^{n-1}_{\mathbf{AS}_2}$.

We prove correctness by induction on n.

Let $\mathcal{MS}^i_2$ and $\mathcal{M}^i_2$ be the partial materialization structure and the partial model respectively, computed in steps $0 \cdots i$.

– $\mathcal{MS}^0_2 = \emptyset$, and $\mathcal{M}^0_2 = \mathcal{M}_1$, by definition. Thus, no dercando literals belong to either of them, and the property holds trivially.
– *Inductive hypothesis:* $(\texttt{dercando}(tuple), \{\ldots, r, \ldots\}) \in \Phi^n_{\mathbf{AS}_2}$ iff $\texttt{dercando}(tuple) \in T^n_{\mathbf{AS}_2}$ and r supports the truth of $\texttt{dercando}(tuple)$ in $T^n_{\mathbf{AS}_2}$.
  Thesis: $(\texttt{dercando}(tuple'), \{\ldots, r', \ldots\}) \in \Phi^{n+1}_{\mathbf{AS}_2}$ iff $\texttt{dercando}(tuple') \in T^{n+1}_{\mathbf{AS}_2}$, and r' supports the truth of $\texttt{dercando}(tuple')$ in $T^{n+1}_{\mathbf{AS}_2}$.
  Let $(\texttt{dercando}(tuple'), \{\ldots, r', \ldots\}) \in \Phi^{n+1}_{\mathbf{AS}_2} \setminus \Phi^n_{\mathbf{AS}_2}$.
  By construction, it must be $\texttt{dercando}(tuple') = head(r')\theta$, for some grounding substitution θ such that the existential closure of $body(r')\theta$ is satisfied in $V_{r'} \cup \Pi_1(\mathcal{MS}^n_2)$. This holds iff the same existential closure is satisfied in $\mathcal{M}^n_2$ (inductive hypothesis), iff $\texttt{dercando}(tuple') \in T^{n+1}_{\mathbf{AS}_2}$, by definition of $T_{\mathbf{AS}_2}$. Thus, r' supports the truth of $\texttt{dercando}(tuple')$ in $T^{n+1}_{\mathbf{AS}_2}$.

j = 3) The proof is similar to the case **j = 1**. □

From Theorem 4 and Definition 5.2, we can derive the following corollaries.

COROLLARY 5.1. *Let* **AS** *be an authorization specification and let* $\mathcal{MS}_i$ *be the materialization structure for its ith stratum,* $\mathbf{AS}_i$. *Then,*

$$\bigcup_{i=0,\ldots,3} \Pi_1(\mathcal{MS}_i)$$

is the unique stable model of $\mathbf{AS}_0 \cup \mathbf{AS}_1 \cup \mathbf{AS}_2 \cup \mathbf{AS}_3$.

COROLLARY 5.2. *Let* **AS** *be an authorization specification and let* $\mathcal{MS}_i$ *be the materialization structure for its ith stratum,* $\mathbf{AS}_i$. *Then,*

$$(\cup_{i=0,\ldots,3}\Pi_1(\mathcal{MS}_i)) \cup \{\texttt{do}(o, s, -a) \mid \texttt{do}(o, s, +a) \notin \Pi_1(\mathcal{MS}_3)\} \cup \Pi_1(\mathcal{MS}_5)$$

is the unique stable model of **AS**.

PROOF. For the first four strata, the result is given by the above corollary. For stratum $\mathbf{AS}_4$, the result comes immediately from the equivalence of $\texttt{do}(o, s, -a)$ and $\neg\texttt{do}(o, s, +a)$. The proof for stratum $\mathbf{AS}_5$ is similar to the case 1 of the proof of the above Theorem 4. □

Before we conclude this section, let us briefly remark on the complexity of steps (1)–(4) for materializing authorization specifications.

PROPOSITION 5.1 (COMPLEXITY OF MATERIALIZATION). *Suppose* **AS** *is an authorization specification whose base relations predicates (*`hie,rel,done`*) are all part of the input. Our materialization procedure has polynomial data complexity.*

PROOF. Steps (0)–(3), and step (5), after the rewriting of integrity rules, materialize a *stratified* logic program [Apt et al. 1988] whose unique stable model computation is well-known to have polynomial time data-complexity [Berman et al. 1995]. The rewriting of the rules, in turn, is computed in polynomial time. No operation is performed on $\mathbf{AS}_4$, thus step (4) does not have any impact on the complexity of the process, and hence the result follows. □

PROPOSITION 5.2 (COMPLEXITY OF COMPUTING THE MODEL). *Suppose* **AS** *is an authorization specification whose base relations predicates (*`hie,rel,done`*) are all part of the input. The computation of the stable model of* **AS** *has polynomial data complexity.*

PROOF. The model of strata (0)–(3), and of stratum (5) is computed in polynomial time, as stated by the above property. So we merely need to show that the model of $\mathbf{AS}_4$ can be computed in polynomial time. This is immediate because for each atom `do(o,s,−a)`, we evaluate `do(o,s,+a)` against those generated in steps (0)–(3) and reverse the truth value. There are polynomially many ground atoms of the form `do(o,s,−a)`, and evaluating each of the queries `do(o,s,+a)` against the view computed in steps (0)–(3) has polynomial data complexity, and hence the result follows. □

5.3 Maintaining Materialized Views with Updates to the Authorization Specification

This section discusses the problem of maintaining the materialization of authorization specification upon changes. Changes can be introduced at any of **AS**'s strata and can be due to changes in data system hierarchies, relationships, or history table ($\mathbf{AS}_0$), as well as due to modifications in the authorization ($\mathbf{AS}_1$), derivation ($\mathbf{AS}_2$), and decision ($\mathbf{AS}_3$) views. From the stratification of the program, we are guaranteed that changes to a stratum $\mathbf{AS}_i$ cannot affect strata below it. The materialization update process exploits this property by incrementally determining the possible changes to the materialization of $\mathbf{AS}_i$ and, iteratively, their effects on the materialization of stratum $i + 1$.

We distinguish the following types of updates to the authorization specification:

–New facts or new rules are inserted into the authorization specification.

–Facts or rules are deleted from the authorization specification.

We do not consider updates that modify already existing facts and rules. This is not a restriction, since such modifications can be realized in terms of deletions and insertions.

5.3.1 *Handling the insertion of facts.* When a new fact is introduced in the authorization specification (stratum $\mathbf{AS}_0$), the materialization structure of every stratum might need to be modified. In principle, the entire authorization specification might change; in practice, it is often the case that the new fact does not have any impact on the extension of any derived predicate. This happens, for example, when the inserted atom is a history fact whose presence does not fire any derivation rule in addition to those that were already fired before the insertion.

Let us consider the insertion of a base fact δ in a `hie`-, `rel`-, or `done`-relation. The process proceeds stratum by stratum, from 0 to 3, computing the required changes. It stops at the first stratum for which no change is recorded (if the model at stratum i is not affected neither will the models of strata above i).

Since amongst the possible consequences of an update there might be an *integrity violation*, an integrity check is always executed before committing the update. This integrity check is performed by evaluating whether `error` belongs to the authorization model after the insertion.

The insertion of a fact is dealt with as follows. Let $\mathbf{AS}^{old}$ denote the authorization specification before the insertion, and $\mathcal{MS}^{old}$ its materialization. The new materialization $\mathcal{MS}^{new}$ is defined as follows:

Step (0): $\mathcal{MS}_0^{new} = \mathcal{MS}_0^{old} \oplus \{(\delta, _)\}$. If $\mathcal{MS}_0^{new} = \mathcal{MS}_0^{old}$, then terminate the materialization process.

Step (1): Let $\texttt{CANDO}^{new}$ be the set of authorization rules in $\mathbf{AS}_1$ whose body contains at least one literal (either positive or negative) that may be unified with the inserted fact, that is, $\texttt{CANDO}^{new} = \{\texttt{cando}(tuple) \leftarrow L_1 \& \cdots \& L_n$ such that for some $i = 1, \ldots, n$, literal L_i unifies with $\delta\}$.

Intuitively, $\texttt{CANDO}^{new}$ is the subset of $\mathbf{AS}_1$ whose extension is potentially affected by the insertion. We then compute the materialization of these rules against the old ($\mathcal{MS}_0^{old}$) and the new ($\mathcal{MS}_0^{new}$) materializations of the lower stratum and compare them. The materializations are computed as described in step (1) of Section 5.2.

Let $\Delta^{new}_{\mathcal{MS}_1}$ be the materialization of $\texttt{CANDO}^{new}$ evaluated against $\mathcal{MS}_0^{new}$.
Let $\Delta^{old}_{\mathcal{MS}_1}$ be the materialization of $\texttt{CANDO}^{new}$ evaluated against $\mathcal{MS}_0^{old}$.
Compute $\Delta^{+}_{\mathcal{MS}_1} = \Delta^{new}_{\mathcal{MS}_1} \ominus \Delta^{old}_{\mathcal{MS}_1}$, the set of pairs to be added to $\mathcal{MS}_1$.
Compute $\Delta^{-}_{\mathcal{MS}_1} = \Delta^{old}_{\mathcal{MS}_1} \ominus \Delta^{new}_{\mathcal{MS}_1}$, the set of pairs to be removed from $\mathcal{MS}_1$.
Set $\mathcal{MS}_1^{new} = \mathcal{MS}_1^{old} \ominus \Delta^{-}_{\mathcal{MS}_1} \oplus \Delta^{+}_{\mathcal{MS}_1}$.
Let $\mathcal{M}_1^{new} = \Pi_1(\mathcal{MS}_0^{new} \cup \mathcal{MS}_1^{new})$, and let $\mathcal{M}_1^{old} = \Pi_1(\mathcal{MS}_0^{old} \cup \mathcal{MS}_1^{old})$.
If $\mathcal{M}_1^{new} = \mathcal{M}_1^{old}$ terminate the materialization process.

Step (2): Let $\Delta_{\mathcal{M}_1} = (\mathcal{M}_1^{new} \setminus \mathcal{M}_1^{old}) \cup (\mathcal{M}_1^{old} \setminus \mathcal{M}_1^{new})$ be the set of atoms in $\mathcal{M}_1$ whose extension has been changed as a consequence of the update.

Compute $\texttt{DERCANDO}^*$ as the set of rules in $\mathbf{AS}_2$ whose firing is potentially affected by the insertion.[9]

[9]Note that in the definition of $\texttt{DERCANDO}^*$ we must take into account the presence of recursion. In addition to the rules whose body contains a literal defined in the lower strata and whose truth could have changed in the corresponding models, we must also consider those rules that possibly depend

DERCANDO* = {dercando($tuple$) $\leftarrow L_1 \& \cdots \& L_n$ such that for some $i = 1,\ldots,n$, literal L_i unifies with an atom in $\Delta_{\mathcal{M}_1}$ or with the head of a rule in DERCANDO*, or dercando($tuple$) unifies with the head of a rule in DERCANDO*}.

Let $\Delta^{new}_{\mathcal{MS}_2}$ and $\Delta^{old}_{\mathcal{MS}_2}$ be the materializations of DERCANDO* with respect to $\mathcal{M}^{new}_1$ and $\mathcal{M}^{old}_1$, respectively. These materializations can be computed as described in step (3) of Section 5.2.

Compute $\Delta^{+}_{\mathcal{MS}_2} = \Delta^{new}_{\mathcal{MS}_2} \ominus \Delta^{old}_{\mathcal{MS}_2}$, the set of new derivations made possible by the insertion of δ.

Compute $\Delta^{-}_{\mathcal{MS}_2} = \Delta^{old}_{\mathcal{MS}_2} \ominus \Delta^{new}_{\mathcal{MS}_2}$, the set of derivations blocked by the insertion of δ.

Set $\mathcal{MS}^{new}_2 = \mathcal{MS}^{old}_2 \ominus \Delta^{-}_{\mathcal{MS}_2} \oplus \Delta^{+}_{\mathcal{MS}_2}$.

Let $\mathcal{M}^{new}_2 = \Pi_1(\mathcal{MS}^{new}_0 \cup \mathcal{MS}^{new}_1 \cup \mathcal{MS}^{new}_2)$, and let $\mathcal{M}^{old}_2 = \Pi_1(\mathcal{MS}^{old}_0 \cup \mathcal{MS}^{old}_1 \cup \mathcal{MS}^{old}_2)$.

If $\mathcal{M}^{new}_2 = \mathcal{M}^{old}_2$ terminate, the materialization process.

Step (3): Derived predicates defined in stratum $\mathbf{AS}_3$ are nonrecursive. Thus, we can follow the same technique discussed in step (1), referring to the facts in the already computed sets $\mathcal{M}^{new}_2$ and $\mathcal{M}^{old}_2$, to evaluate the derivation increments and decrements.

Step (4) (Integrity check): Let $\Delta_{\mathcal{M}_3} = (\mathcal{M}^{new}_3 \setminus \mathcal{M}^{old}_3) \cup (\mathcal{M}^{old}_3 \setminus \mathcal{M}^{new}_3)$.

Let ERROR* be the set of error rules whose body contains a literal which unifies with some atom in $\Delta_{\mathcal{M}_3}$, or with some do($o,s,-a$) atom, such that do($o,s,+a$) belongs to $\Delta_{\mathcal{M}_3}$.

Rewrite the rules in ERROR*, replacing any do($o,s,-a$) atom with $\neg$do($o,s,+a$), and evaluate them against $\mathcal{M}^{new}_3$.

Some comments on the effectiveness of the method discussed above are in order. First, we comment on the *frequency* of the updates. While updates to hie and rel predicates are not likely to be frequent, updates to done may be very frequent (as a matter of fact the main reason for materializing the model of the specification is that access requests are far more frequent than administrative requests). One may assert that, given that the history is recorded using the done predicate, any access to the system on which the authorization specification is defined may result in an insertion of a new done fact, implying a change to the specification and, as a consequence, changes to the specifications may be far greater in number than access requests.

A careful examination shows that insertion of done facts is a particular change to an authorization specification that has limited effect on its unique stable model. We can take this into account and treat insertion of new done predicates with an “ad hoc” strategy that addresses the following two aspects:

– *How does the granularity of the time component in* done *predicates relate to the time granularity of the system that records history facts?* If a number of

on the update through recursion. To perform this dependency check, we refer to the original rules in $\mathbf{AS}_2$. Every time $\mathbf{AS}_2$ is materialized, a new rewritten version of the potentially affected rules is constructed, since the materialization of the conjunction of nonrecursive literals might change (see Section 5.2).

actions whose system times are necessarily distinct, collapse into the same done fact, then it may very well be the case that the update process stops at step 0, since an already existing fact is inserted. This can dramatically reduce the complexity of the method due to the insertion of history facts.

–*How relevant is the time component in history facts?* The time component is relevant for history facts only in those cases in which they can be unified with some literal done(_, _, _, _, t) in a derived rule whose body also contains a test on the time term t. Indeed, time is not present as an argument of the derived predicates, thus the only role it can play is to allow/block the firing of some derivation rule through the comparison of different time instants in a given rule.

We can then exploit this characteristic as follows. When a history fact is introduced, we first check whether the time term can be relevant to the firing of any derivation rule. If the answer is yes, then the described method is followed to propagate the update to the upper strata. If the answer is not, the change does not affect the views at the upper strata and therefore no further propagation action needs to be taken. Since we expect that only a small subset of the history-based rules will contain comparison of time terms, this approach "effectively reduces" the complexity of the process.

–Of course, the suggested strategies can be combined, if the granularity of the history facts is different from the granularity of the system.

In the same vein, some optimizations can be incorporated into the integrity check phase. Though we have described integrity checking as though it is a single step enforced at the completion of the materialization process, its evaluation could easily be executed incrementally, terminating the process at the first strata which generates an error. Intuitively, we could imagine error rules partioned so that an integrity rule is associated with the highest strata of literals in its body. After a stratum of the authorization base has been re-materialized, the integrity rules associated with that stratum can be evaluated. It is easy to see that while the evaluation of the integrity rules so partitioned is equivalent to that performed in a single final step; the incremental approach, by evaluating integrity rules as soon as possible in the materialization process, avoids unnecessary computation. Notice that the incremental approach is convenient if the SSO is interested in avoiding inconsistent updates, and stopping the update process as soon as an inconsistency arises. If the SSO wants to proceed until the end, to know, for example, what are all the consistency rules that would be violated by a given update, the two approaches are equivalent.

THEOREM 4 (CORRECTNESS OF THE UPDATE PROCEDURE). *Let* **AS** *be an authorization specification,* $\mathcal{MS}^{old}$ *be the materialization structure that correctly models* **AS**, *and* δ *be a base fact to be inserted. The update procedure transforms* $\mathcal{MS}^{old}$ *into* $\mathcal{MS}^{new}$ *such that* $\mathcal{MS}^{new}$ *correctly models* $\mathbf{AS} \cup \{\delta\}$.

PROOF. We prove the correctness of each stratum (i.e., each step) of the algorithm.

Step (0): Correctness of $\mathcal{MS}_0^{new}$ holds trivially.

Step (1): We have to show that for every ground fact $\texttt{cando}(\mathit{tuple})$, it holds that $\texttt{cando}(\mathit{tuple}) \in \mathcal{M}(\mathbf{AS} \cup \{\delta\})$ iff, for every rule r in $\mathbf{AS}_1$ such that $\texttt{cando}(\mathit{tuple}) = \mathit{head}(r)\theta$ and $\mathit{body}(r)\theta$ is true in $\mathcal{M}(\mathbf{AS} \cup \{\delta\})$ for some substitution θ, there is a pair $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\})$ in $\mathcal{MS}^{new}$.

From the stratification of the program, $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \mathcal{MS}^{new}$ iff $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \mathcal{MS}_1^{new}$. Then, we only consider strata $\mathbf{AS}_0$ and $\mathbf{AS}_1$.

Let r be any rule in $\mathbf{AS}_1$ such that $\texttt{cando}(\mathit{tuple}) = \mathit{head}(r)\theta$, for some variable substitution θ. $\mathit{body}(r)\theta$ is true in $\mathcal{M}(\mathbf{AS} \cup \{\delta\})$ iff it is true in $\mathcal{M}(\mathbf{AS}_0 \cup \{\delta\})$. We distinguish two cases:

(a) No literal in $\mathit{body}(r)\theta$ unifies with δ.

Then, $\mathit{body}(r)\theta$ is true in $\mathcal{M}(\mathbf{AS}_0 \cup \{\delta\})$ iff it is true in $\mathcal{M}(\mathbf{AS}_0)$ iff $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \mathcal{MS}_1^{old}$, where the last step comes from the hypothesis that $\mathcal{MS}^{old}$ correctly models $\mathbf{AS}$. By construction, $r \notin \texttt{CANDO}^{new}$, thus neither $\Delta^{new}_{\mathcal{MS}_1}$ nor $\Delta^{old}_{\mathcal{MS}_1}$ contains $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\})$. Hence, from the definition of operators $\oplus$ and $\ominus$, we have $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \mathcal{MS}_1^{new} = \mathcal{MS}_1^{old} \ominus \Delta^-_{\mathcal{MS}_1} \oplus \Delta^+_{\mathcal{MS}_1}$.

(b) Some literal in $\mathit{body}(r)\theta$ unifies with δ.

Three cases are possible, depending on whether the literals that unify with δ are all positive, all negative, or some are positive and some negative.

Case (1): Only positive literals in $\mathit{body}(r)\theta$ unify with δ.
Assume $\mathit{body}(r)\theta$ is not true in $\mathbf{AS}_0$ (otherwise, correctness holds trivially). Since, by hypothesis, $\mathcal{MS}^{old}$ is correct, then $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \mathcal{MS}_1^{old}$. If $\mathit{body}(r)\theta$ is true in $\mathcal{M}(\mathbf{AS}_0 \cup \{\delta\})$, it must be $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \Delta^{new}_{\mathcal{MS}_1}$. By construction and by definition of $\oplus$ and $\ominus$, $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \Delta^+_{\mathcal{MS}_1}$ and $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \Delta^-_{\mathcal{MS}_1}$, thus $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \mathcal{MS}_1^{new} = \mathcal{MS}_1^{old} \ominus \Delta^-_{\mathcal{MS}_1} \oplus \Delta^+_{\mathcal{MS}_1}$. If $\mathit{body}(r)\theta$ is not true in $\mathcal{M}(\mathbf{AS}_0 \cup \{\delta\})$, it was not true in $\mathcal{M}(\mathbf{AS}_0)$. From the correctness of $\mathcal{MS}_0$, $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \mathcal{MS}_1^{old}$. Moreover, it is not added to the materialization structure by the update procedure. Thus, $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \mathcal{MS}_1^{new}$, which proves the correctness of the procedure.

Case (2): Only negative literals in $\mathit{body}(r)\theta$ unify with δ.
The existential closure of $\mathit{body}(r)\theta$ is not satisfied in $\mathcal{M}(\mathbf{AS}_0 \cup \{\delta\})$, thus $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \mathcal{MS}_1^{new}$. It can be the case, however, that there exists $(\texttt{cando}(\mathit{tuple}), S) \in \mathcal{MS}_1^{new}$, with $r \notin S$. Hence, if the existential closure of $\mathit{body}(r)\theta$ was not satisfied in $\mathcal{M}(\mathbf{AS}_0)$, correctness holds trivially: r does not appear in any set supporting $(\texttt{cando}(\mathit{tuple})$ in $\mathcal{MS}_1^{old}$, and it is not introduced in $\mathcal{MS}_1^{new}$, since by construction $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \Delta^+_{\mathcal{MS}_1}$. If the existential closure of $\mathit{body}(r)\theta$ was satisfied in $\mathcal{M}(\mathbf{AS}_0)$, $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \in \Delta^-_{\mathcal{MS}_1}$ and $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \Delta^+_{\mathcal{MS}_1}$. Thus, from the definition of $\oplus$ and $\ominus$, $(\texttt{cando}(\mathit{tuple}), \{\ldots, r, \ldots\}) \notin \mathcal{MS}_1^{new}$; which proves the correctness of the procedure.

Case (3): Both positive and negative literals in $body(r)\theta$ unify with δ. The existential closure of $body(r)\theta$ cannot be satisfied, thus the procedure is trivially correct.

Step (2): The proof of correctness of step (2) is slightly different from that of step (1), because of the possible positive recursion of dercando-rules.

The algorithm partitions $\mathbf{AS}_2$ in two subsets of rules, DERCANDO* and ($\mathbf{AS}_2 \setminus$ DERCANDO*) such that $\mathcal{M}(\mathbf{AS}_2) = \mathcal{M}(\texttt{DERCANDO}^*) \cup \mathcal{M}(\mathbf{AS}_2 \setminus \texttt{DERCANDO}^*)$.

This equality comes from the fact that, by definition of DERCANDO*, literals defined in $\mathbf{AS}_2 \setminus$ DERCANDO* do not unify with literals appearing in either the head or the body of the rules in DERCANDO*. Thus, the two subsets are clearly independent of each other.

By construction, rules in ($\mathbf{AS}_2\setminus$DERCANDO*) do not depend, either directly or indirectly, on δ. Thus, from the correctness of $\mathcal{MS}^{old}$, for any such rules r and any ground instance dercando(*tuple*) of $head(r)$ (via mgu θ), we have (dercando(*tuple*), $\{\ldots, r, \ldots\}) \in \mathcal{MS}^{old}$ iff $body(r)\theta$ is true in $\mathcal{MS}_2^{old}$, iff $body(r)\theta$ is true in $\mathcal{MS}_2^{new}$. Thus, for these rules the algorithm is correct.

Now, consider any rule r in DERCANDO* and any ground fact dercando(*tuple*) $=$ $head(r)\theta$. By construction and from the definition of $\oplus$ and $\ominus$, (dercando(*tuple*), $\{\ldots, r, \ldots\}) \in \mathcal{MS}_2^{new}$ iff either (dercando(*tuple*), $\{\ldots, r, \ldots\}) \in \mathcal{MS}_2^{old}$ and (dercando(*tuple*), $\{\ldots, r, \ldots\}) \notin \Delta^-_{MS_2}$, or (dercando(*tuple*), $\{\ldots, r, \ldots\}) \notin \mathcal{MS}_2^{old}$ and (dercando(*tuple*), $\{\ldots, r, \ldots\}) \in \Delta^+_{MS_2}$. In both cases, from Theorem 4, (dercando(*tuple*), $\{\ldots, r, \ldots\}) \in \mathcal{MS}_2^{new}$ iff $body(r)\theta$ is true in $\mathcal{M}(\mathbf{AS}_2)$, which proves the correctness of the procedure.

Step (3): Correctness of the third step can be shown using arguments similar to those used to prove the correctness of step (1), by considering $\mathbf{AS}_3$ as partitioned in two subsets, one containing the rules that possibly depend on the update and the other containing the rules that do not.

Step (4) (Integrity Check): Correctness of the integrity check phase can be shown using arguments similar to those used to prove the correctness of step (1), by considering $\mathbf{AS}_5$ as partitioned in two subsets, one containing the rules that possibly depend on the update and the other containing updates that do not. □

PROPOSITION 5.3 (COMPLEXITY OF THE UPDATE PROCEDURE). *Suppose that* **AS** *is an authorization specification whose base relations predicates (*hie,rel,done*) are all part of the input, and that A is any atom being inserted. Our update procedure has polynomial data complexity.*

PROOF. Step (0) is clearly executable in polynomial time. In step (1), finding all atoms in rule bodies that unify with an inserted fact is polynomial (it only involves a linear scan of the rules and unification is known to be doable in polynomial time [Martelli and Montanari 1982]). Computing the (new) materialization of these rules is also polynomial by our previous complexity result on materialization of authorization rules. Step (2) is also polynomial for the same reason. Step (3) is polynomial for the same reason step (1) is polynomial. Integrity check is polynomial since the rewriting of rules

is polynomial and so is their evaluation, for the same reason step (1) is polynomial. □

5.3.2 *Handling the Insertion of Rules.* The iterative process we have introduced to handle the insertion of base facts in the materialized authorization specification is in fact a general method that can be applied to handle the insertion of rules as well as the deletion of both facts and rules.

5.3.2.1 INSERTION OF `cando` AND `do` RULES. When a rule is inserted in stratum $\mathbf{AS}_1$, (respectively, stratum $\mathbf{AS}_3$), that is, in a stratum without recursion, the process in Section 5.3.1 is applied, starting from step (1) (respectively, step (3)), and letting $\texttt{CANDO}^{new}$ (respectively, $\texttt{DO}^{new}$), that is, the sets of rules potentially affected by the insertion, contain exactly the inserted rule. The algorithm checks whether the new rule allows the derivation of new atoms. If the answer is yes, the update to the considered stratum is propagated to the upper levels, otherwise the algorithm stops (successfully).

5.3.2.2 INSERTION OF `dercando` RULES. If a rule is inserted in stratum $\mathbf{AS}_2$, then the update process is started at step (2), letting $\texttt{DERCANDO}^* = \{\texttt{dercando}(tuple) \leftarrow L_1 \& \cdots \& L_n$ such that for some $i = 1, \ldots, n$, literal L_i unifies with the head of the inserted rule or with the head of a rule in $\texttt{DERCANDO}^*\}$. Intuitively, $\texttt{DERCANDO}^*$ contains all the rules that could fire because of the presence of the new rule introduced. Correctness of the approach comes from the fact that recursion can only be positive, thus the immediate consequence operator $\Phi_{\mathbf{AS}_2}$ is monotonic. Hence, the authorizations that become true on the basis of the new rule cannot block any previously firing rule in the same stratum.

It is easy to see that insertion of `cando` and `dercando` rules into an authorization specification preserves the (polynomial) data complexity results we have obtained earlier.

5.3.3 *Handling Deletions.* We distinguish deletion of base facts, deletion of rules in strata that do not allow recursion, and deletion of rules in the potentially recursive stratum.

5.3.3.1 DELETION OF BASE FACTS. To handle the deletion of a base fact δ from the authorization specification, we apply the update method starting from step (0) and letting $\mathcal{MS}_0^{new} = \mathcal{MS}_0^{old} \ominus \{(\delta, _\}$. This step can obviously be executed with polynomial data complexity.

5.3.3.2 DELETION OF `cando` AND `do` RULES. The deletion of a rule from $\mathbf{AS}_1$ (or $\mathbf{AS}_3$) is taken care of with a method that is symmetric to the insertion: the set $\texttt{CANDO}^{new}$ ($\texttt{DO}^{new}$, respectively) containing exactly the removed rule, is considered as the set of rules potentially affected by the update, and its materialization $\Delta^-_{\mathcal{MS}_1}$ ($\Delta^-_{\mathcal{MS}_3}$, respectively) is computed. $\Delta^-_{\mathcal{MS}_i}$ contains the derived atoms that were supported by the removed rule. Note that there is no need to calculate $\Delta^{new}_{\mathcal{MS}_i}$ and $\Delta^+_{\mathcal{MS}_i}$, since $\Delta^+_{\mathcal{MS}_i}$ will always be empty given that removing a rule from $\mathbf{AS}_i$ cannot cause new atoms to be added to MS_i. Thus, $\mathcal{MS}_i^{new} = \mathcal{MS}_i^{old} \ominus \Delta^-_{\mathcal{MS}_i}$ is computed.

If $\Pi_1(\mathcal{MS}_i^{new}) \neq \Pi_1(\mathcal{MS}_i^{old})$, then the control is passed to the next step. Note that it is not necessary to look at $\mathcal{M}_i$ (which also contains atoms in the models of lower strata), since nothing has changed at levels below i. Note also that facts that were supported by the deleted rule will still belong to the model if there are other rules supporting them. A fact is removed from the materialization structure (and hence from the model) only when its set of supporting rules becomes empty.

Deletion of cando and do rules is a polynomial process because computing the materializations $\Delta^-_{\mathcal{MS}_1}$ and $\Delta^-_{\mathcal{MS}_3}$ is polynomial (as materializing a rule has polynomial data complexity by our previous results). Computing $\mathcal{MS}_i^{new} = \mathcal{MS}_i^{old} \ominus \Delta^-_{\mathcal{MS}_i}$ is clearly also polynomial (in fact quadratic in the sizes of the relations involved). Checking if $\Pi_1(\mathcal{MS}_i^{new}) \neq \Pi_1(\mathcal{MS}_i^{old})$ is also polynomial.

5.3.3.3 DELETION OF dercando RULES. The deletion of dercando rules is based on a technique similar to the approach adopted in the corresponding step of the insertion algorithm. In this case, recursion is taken into account defining a set DERCANDO* as follows ($r \in$ DERCANDO* is the rule to be deleted):

DERCANDO* $= \{r\} \cup \{$dercando(*tuple*) $\leftarrow L_1 \& \cdots \& L_n$, such that for some $i = 1, \ldots, n$ a literal L_i unifies with the head of a rule in DERCANDO*, or dercando(*tuple*) unifies with the head of a rule in DERCANDO*$\}$.

DERCANDO* contains all the rules possibly connected to the deleted rule. For any rule in DERCANDO*, its head unifies with the head of some rule whose firing might depend on the head predicate of the deleted rule. The deletion does not have any effects on the model of $\mathbf{AS}_2 \setminus$DERCANDO*.

Let NEWDERCANDO* $=$ DERCANDO*$\setminus\{r\}$. The two sets DERCANDO* and NEWDERCANDO* will now be materialized to find the impact that the rule r had on the old materialization, $\mathcal{MS}_2$. The materialization of DERCANDO* includes all pairs $(head(r)\theta, r)$ for all θ such that $head(r)\theta \in \mathcal{MS}_2^{old}$. Such pairs will not belong to the materialization of NEWDERCANDO* (remember that r is not in NEWDERCANDO*). However, note that it may not be enough to materialize DERCANDO* as it is, since the truth value of $body(r)$ in DERCANDO* may or not be true, something which will affect the presence of $head(r)$ in the materialization. We must therefore evaluate DERCANDO* against the materializations of the lower strata $\mathcal{MS}_0$ and $\mathcal{MS}_1$, with the addition of $\{(head(r)\theta, r)\}$, to make sure that $\mathcal{MS}_{\text{DERCANDO}^*}$ includes all atoms that are produced as a result of the truth of $head(r)\theta$.

The materializations $\mathcal{MS}_{\text{DERCANDO}^*}$ (with the above adjustment) and $\mathcal{MS}_{\text{NEWDERCANDO}^*}$ of DERCANDO* and NEWDERCANDO*, respectively, are computed. We compute the set $R = \mathcal{MS}_{\text{DERCANDO}^*} \ominus \mathcal{MS}_{\text{NEWDERCANDO}^*}$ of pairs that are in $\mathcal{MS}_{\text{DERCANDO}^*}$ but not in $\mathcal{MS}_{\text{NEWDERCANDO}^*}$, and finally, $\mathcal{MS}_2^{new} = \mathcal{MS}_2 \ominus R$.

The complexity of computation DERCANDO* is obviously quadratic in the size of our authorization specification. By our previous complexity result on materialization, we can see that the materializations $\mathcal{MS}_{\text{DERCANDO}^*}$ and $\mathcal{MS}_{\text{NEWDERCANDO}^*}$ can be computed with polynomial data complexity. Finally, computing $\mathcal{MS}_{\text{DERCANDO}^*} \ominus \mathcal{MS}_{\text{NEWDERCANDO}^*}$ is also immediately seen to be polynomial.

Example 5.1. Assume the following five rules are given:

$\mathtt{r}_1$: dercando(tuple1) $\leftarrow$ dercando(tuple2)

r_2: dercando(tuple1) ← non-recursive-body
r_3: dercando(tuple3) ← dercando(tuple1)
r_4: dercando(tuple4) ← dercando(tuple2)
r_5: dercando(tuple2) ← non-recursive-body

Assume that the existential closures of the bodies of nonrecursive rules are all satisfied, and that tuple1, tuple2, tuple3 are ground, distinct tuples. Thus,

$$\mathcal{MS}_2 = \{(\texttt{dercando(tuple2)}, \{r_5\}), (\texttt{dercando(tuple1)}, \{r_1, r_2\}), (\texttt{dercando(tuple3)}, \{r_3\}), (\texttt{dercando(tuple4)}, \{r_4\})\}$$

Suppose the SSO requests that rule r_1 be removed.

$\texttt{DERCANDO}^* = \{r_1, r_2, r_3\}$, and $\texttt{NEWDERCANDO}^* = \{r_2, r_3\}$.
$\mathcal{MS}_{\texttt{DERCANDO}^*} = \{(\texttt{dercando(tuple1)}, \{r_1, r_2\}), (\texttt{dercando(tuple3)}, \{r_3\})\}$
$\mathcal{MS}_{\texttt{NEWDERCANDO}^*} = \{(\texttt{dercando(tuple1)}, \{r_2\}), (\texttt{dercando(tuple3)}, \{r_3\})\}$
$R = \mathcal{MS}_{\texttt{DERCANDO}^*} \ominus \mathcal{MS}_{\texttt{NEWDERCANDO}^*} = \{(\texttt{dercando(tuple1)}, \{r_1\})$.

Thus, only $(\texttt{dercando(tuple1)}, \{r_1\})$ is removed from the materialization structure, which then becomes:

$$\mathcal{MS}_2^{new} = \{(\texttt{dercando(tuple2)}, \{r_5\}), (\texttt{dercando(tuple1)}, \{r_2\}), (\texttt{dercando(tuple3)}, \{r_3\}), (\texttt{dercando(tuple4)}, \{r_4\})\}$$

6. RELATED WORK

Other researchers have investigated the problem of enforcing different access control policies within a single data system. In this direction, the recent implementations of the microkernel-based operating systems (e.g., Trusted Mach [Branstad et al. 1989], Synergy [Saydjari et al. 1993], and Distributed Trusted Operating System (DTOS) [Fine and Minear 1993]) cleanly separate the policy enforcement from the policy decision. A *policy-neutral security server* that is inside the microkernel is responsible for the enforcement of the policy decision; the policy decision is left to a *security server* that is outside the microkernel. Since the computation of access decisions based on a particular policy is separate from the enforcement mechanism, it is possible to implement different policies on the microkernels by simply inserting the right security server. These proposals, although moving in the direction of supporting multiple policies, still require the system to maintain different security servers, and replace the one to be applied as needed. Moreover, they cannot capture access control restrictions different from those based on explicit authorizations or subject classifications.

Woo and Lam [1993] propose a logic language for the specification of authorizations. Their proposal does not investigate specifically the type of predicates and constraints (e.g., derivation, conflict resolution, etc.) usually found in access control systems but, by using a very general language, which has almost the same expressive power of first-order logic, gives the possibility of expressing them. The drawback of such an approach is that the trade-off between expressiveness and efficiency seems to be strongly unbalanced. In particular, the lack of any restrictions whatsoever on the language results in the specification of models that may not even be decidable and therefore will not be implementable.

As a matter of fact, Woo and Lam's approach is based on truth in extensions of arbitrary default theories, which is known, even in the propositional case to be NP-complete, and in the first-order case, is worse than undecidable, as shown by Gottlob [1992]. By contrast, our approach identifies a polynomial time (in fact quadratic time) data complexity fragment of default logic. Therefore, in effect, our approach is effectively implementable, while the one in Woo and Lam [1993] is not. As illustrated in the paper, our polynomial-time fragment is able to establish all the well-known database security policies that we have come across. While this statement does not necessarily cover all possible security policies that may be devised, it certainly covers the commonly used policies of today.

Another approach proposing a logic language for the specification of access permissions and restrictions to be enforced is presented in Bertino et al. [1996]. The approach, however, is mainly devoted to the consideration of temporal constraints that may need to be enforced on the authorizations and time reasoning and does not consider other policy issues. For instance, no relationships between the components of the data systems are considered and possible conflicts are solved according to the basic denials-take-precedence policy.

A first proposal towards the specification of a more powerful and complete, yet actually computable, language for the specification of authorizations was made by us in Jajodia et al. [1997a; 1997b]. There, we presented some of the concepts that are the basis of our current model. However, no formalization of the data system that would allow precise definition of the model behavior was presented. Also, implementation issues were not investigated.

Starting from the concepts presented in Jajodia et al. [1997a; 1997b], Bertino et al. [1998] propose a formal language for the specification and enforcement of authorizations. Although their proposal is based on a language more general than ours, it cannot capture all the different policies and constraints expressible in our simpler and efficiently implementable framework. For instance, the integrity (our error rule) or history constraints (our done rule) cannot be expressed. This last aspect, in particular, makes it impossible for the model in Bertino et al. [1998] to capture any form of dynamic constraints, such as separation of duty or Chinese Wall policies. Moreover, in Bertino et al. [1998], conflict resolution and decision policies cannot be expressed in the language but are built into the model, thus going back to a more rigid framework that does not easily accommodate different protection requirements but forces their specification in terms of the policy enforced by the system. The generality of the language also allows for specifications having more than a stable model, whose meaning and acceptability in an access control framework (where access to be allowed or not must be clearly determined and no ambiguity should be present) is doubtful. Finally, the proposal in Bertino et al. [1998] also suffers from the same drawback as Woo and Lam [1993]; while our approach is effectively implementable, the one in Bertino et al. [1998] is not.

7. CONCLUSIONS

Over the years, researchers have proposed a vast variety of access control policies and models. However, most practical systems that have been deployed have

traditionally chosen to implement just one policy. As a consequence, applications constructed on top of such systems, as well as the users of such systems, are forced to use this implemented policy. The aim of this paper is to move away from this rigidity and, instead, to provide the application developer with a host of policy options. To achieve the desired flexibility, we have proposed a flexible authorization framework, based on a powerful yet simple language, through which a system security officer may specify access control requirements through explicit authorizations, together with derivation, conflict resolution and decision strategies. Different strategies may be applied to different users, groups, objects, or roles, according to the needs of the security policy. The framework also allows the enforcement of rich constraints that are generally required in real world applications, but very seldom supported by the access control systems. The major advantage of our approach is that it can be used to specify different access control policies that can all coexist in the same system and that can be enforced by the same security server.

Our paper leaves room for further work. A first issue we plan to investigate concerns administrative policies. In this paper, we have made the assumption that all specifications are stated by the System Security Officer. The model can be extended to include administrative policies that regulate insertions of the different rules by the users. We also plan to investigate how well our model can represent and enforce complex security policies of different organizations, such as financial or health-care institutions.

ACKNOWLEDGMENTS

We are grateful to Åsa Hagström, Duminda Wijesekera, Francesco Parisi-Presicce, and the anonymous referees for their careful reading of the paper, constructive criticisms, and useful suggestions, that led to many improvements to the paper.

REFERENCES

APT, K., BLAIR, H., AND WALKER, A. 1988. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed., Morgan-Kaufmann, San Mateo, Calif.

BARAL, C. AND SUBRAHMANIAN, V. 1992. Stable and extension class theory for logic programs and default theories. *J. Automat. Reas. 8*, 345–366.

BERMAN, K., SCHLIPF, J., AND FRANCO, J. 1995. Computing the well-founded semantics faster. In *Proceedings of the 3rd International Workshop on Logic Programming and Nonmonotonic Reasoning*, A. N. W. Marek and M. Truszczynski, Eds., (Lexington, Ky., June). pp. 113–126.

BERTINO, E., BETTINI, C., FERRARI, E., AND SAMARATI, P. 1996. A temporal access control mechanism for database systems. *IEEE Trans. Knowl. Data Eng. 8*, 1, 67–80.

BERTINO, E., BUCCAFURRI, F., FERRARI, E., AND RULLO, P. 1998. An authorizations model and its formal semantics. In *Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS'98)* (Louvaine-Le-Neuve, Belgium).

BERTINO, E., JAJODIA, S., AND SAMARATI, P. 1999. A flexible authorization mechanism for relational data management systems. *ACM Trans. Inf. Syst. 17*, 2, 101–140.

BERTINO, E., SAMARATI, P., AND JAJODIA, S. 1993. Authorizations in relational database management systems. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, VA. Nov. 3–5). ACM, New York, pp. 130–139.

BRANSTAD, M., TAJALLI, H., MAYER, F., AND DALVA, D. 1989. Access mediation in a message passing kernel. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 66–72.

BREWER, D. F. C. AND NASH, M. J. 1989. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 215–228.

BRÜGGEMANN, H. H. 1992. Rights in an object-oriented environment. In *Database Security, V: Status and Prospects*, North-Holland, Amsterdam, The Netherlands, pp. 99–115.

CASTANO, S., FUGINI, M., MARTELLA, G., AND SAMARATI, P. 1995. *Database Security*. Addison-Wesley, Reading, Mass.

DENNING, D. E., LUNT, T., SCHELL, R., HECKMAN, M., AND SHOCKLEY, S. 1987. Secure distributed data view (Sea View) — the Sea View formal security policy model. Tech. rep. SRI International, Menlo Park, Calif.

FINE, T. AND MINEAR, S. E. 1993. Assuring distributed trusted mach. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 206–218.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming* (Seattle, Wash.). pp. 1070–1080.

GOTTLOB, G. 1992. Complexity results for nonmonotonic logics. *J. Logic Comput. 2*, 3, 397–425.

JAJODIA, S., SAMARATI, P., AND SUBRAHMANIAN, V. 1997a. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 94–107.

JAJODIA, S., SAMARATI, P., SUBRAHMANIAN, V., AND BERTINO, E. 1997b. A unified framework for enforcing multiple access control policies. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (Tucson, AZ, May 13–15). ACM, New York, pp. 474–485.

JONSCHER, D., AND DITTRICH, K. R. 1996. Argos — A configurable access control system for interoperable environments. In *Database Security IX: Status and Prospects*, S. A. D. D. L. Spooner and J. E. Dobson, Eds., Chapman & Hall, London, England, pp. 43–60.

LLOYD, J. W. 1987. *Foundations of Logic Programming*. Springer-Verlag, New York.

LUNT, T. F. 1989. Access control policies for database systems. In *Database Security II: Status and Prospects*, C. E. Landwehr, Ed., North-Holland, Amsterdam, The Netherlands, pp. 41–52.

MAREK, W. AND SUBRAHMANIAN, V. 1992. The relationship between stable, supported, default and auto-epistemic semantics for general logic programs. *Theoret. Comput. Sci. 103*, 365–386.

MARTELLI, A. AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Prog. Lang. Syst. 4*, 2, 258–282.

PRZYMUSINSKI, T. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases*, J. Minker, Ed., Morgan-Kaufmann, San Mateo, Calif., pp. 193–216.

RABITTI, F., BERTINO, E., KIM, W., AND WOELK, D. 1991. A model of authorization for next-generation database systems. *ACM Trans. Data. Syst. 16*, 1, 89–131.

REITER, R. 1980. A logic for default reasoning. *Artif. Int. 13*, 81–132.

SAYDJARI, O. S., TURNER, S. J., PEELE, D. E., FARRELL, J. F., LOSCOCCO, P. A., KUTZ, W., AND BOCK, G. L. 1993. Synergy: A distributed, microkernel-based security architecture, version 1.0. Tech. rep. National Security Agency, Ft. George G. Meade, Md.

SHEN, H. AND DEWAN, P. 1992. Access control for collaborative environments. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. ACM, New York, pp. 51–58.

TARSKI, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math. 5*, 285–309.

VAN GELDER, A. 1989. The alternating fixpoint of logic programs with negation. In *Proceedings of the 8th ACM SILACT-SICMOO-SILART Symposium on Principles of Database Systems* (Philadelphia, Pa., Mar. 29–31). ACM, New York, pp. 1–10.

WOO, T. Y. C. AND LAM, S. S. 1993. Authorizations in distributed systems: A new approach. *Journal of Computer Security 2*, 2,3.

Received March 1999; revised October 2000; accepted November 2000