A Logical Language for Expressing Authorizations*

Sushil Jajodia
Center for Secure Information Systems and
ISSE Dept., George Mason University
Fairfax, VA 22030-4444, USA
email: jajodia@gmu.edu

Pierangela Samarati
Dipartimento di Scienze dell'Informazione
Università di Milano
20135 Milano, Italy
email: samarati@dsi.unimi.it

V. S. Subrahmanian
Department of Computer Science
University of Maryland
College Park, MD 20742, USA
email: vs@cs.umd.edu

Abstract

A major drawback of existing access control systems is that they have all been developed with a specific access control policy in mind. This means that all protection requirements (i.e., accesses to be allowed or denied) must be specified in terms of the policy enforced by the system. While this may be trivial for some requirements, specification of other requirements may become quite complex or even impossible. The reason for this is that a single policy simply cannot capture different protection requirements users may need to enforce on different data.

In this paper we take a first step towards a model able to support different access control policies. We propose a logical language for the specification of authorizations on which such a model can be based. The language allows users to specify, together with the authorizations, the policy according to which access control decisions are to be made. Policies are expressed by means of rules which enforce derivation of authorizations, conflict resolution, access control, and integrity constraint checking. We illustrate the power of our language by showing how different constraints that are sometimes required, but very seldom supported by existing access control systems, can be represented in our language.

1 Introduction

Access control policies govern the access of users to information on the basis of the users' identity and a collection of rules (or *authorizations*) which determine, for any user and any object in the system, the types of accesses (e.g., read, write, or execute) the user is allowed on the object. There are many different choices of access control policies [8]. In a *closed policy*, all accesses that are to be allowed (we call these *positive* authorizations) must be specified. The request of a user to access an object is checked against the specified authorizations; if there exists an authorization stating that the user can access the object in the specific mode, the access is granted, otherwise it is denied. Alternatively, in an *open policy*, all accesses to be denied (we call these *negative* authorizations) have to be fully specified; users are allowed all those accesses that they have not been explicitly denied. Some policies permit specification of both positive and negative authorizations. Policies which permit conflicting authorizations to be specified on the same data object provide an *overriding* (e.g., *denials-take-precedence*) rule stating how such conflicts are to be resolved.

While an access control policy defines what is authorized, an access control mechanism implements the policy by ensuring that all accesses are in accordance with the underlying policy. In studying access control, it is useful to separate policy and mechanism for several reasons: First, we divide the problem into two subproblems, each of which can be addressed separately. Second, it is possible to have different mechanisms that enforce the same policy. Third, it is possible have mechanisms that are capable of implementing multiple security policies. This last feature is very important from a practical standpoint. If we have a general mechanism that is capable of implementing a number of different security policies, we can replace the security policy without requiring a change in the mechanism.

Unfortunately, almost all access control models and mechanisms fail to maintain this separation, and are tailored specifically to meet the requirements for the closed policy. This rigidity creates a problem when the protection requirement of an application are different from the policy built into the mechanism at hand. In most cases, the only solu-

*The work of Sushil Jajodia was partially supported by National Science Foundation under grants IRI–9633541 and INT–9412507 and by National Security Agency under grants MDA904–96–1–0103 and MDA904–96–1–0104. The research of V.S. Subrahmanian was supported in part by the Army Research Office under grants DAAH-04-95-10174 and DAAH-04-96-10297, by ARPA/Rome Labs contract F4630602-93-C-0241 (Order A716), by an NSF Young Investigator award IRI-93-57756, and by the Army Research Laboratory under Cooperative Agreement DAAL01-96-2-0002 Federated Laboratory ATIRP Consortium.

tion 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.

The recent implementations of the microkernel-based operating systems (e.g., Trusted Mach [4], Synergy [10], and Distributed Trusted Operating System (DTOS) [6]) cleanly separate the policy enforcement from the policy decision. A *policy-neutral security server* which is inside the microkernel is responsible for the enforcement of the policy decision; the policy decision is left to a *security server* which 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.

Although this is a step in the right direction, we go one step beyond this by making it possible to implement different policies without requiring a replacement of even the security server. We present a language that can be used by users to express their security policies. The language comprises different kinds of rules: derivation rules, which allow derivation of implicit authorizations from those explicitly specified and determine the overriding policy to be applied; resolution rules, which regulate how conflicts between authorizations must be resolved; access control rules, which regulate access control policy decisions; and integrity rules, which are used to express different kinds of constraints on the specification and use of authorizations. Different policies can be specified for different objects. For instance, access to an object can be controlled according to the closed policy and access to another object according to the open policy. A nice feature of our approach is that the *same* security server, which evaluates these rules, can support different policies, depending on the specifications.

The first attempt aimed at providing a general framework for expressing authorizations was made by Woo and Lam [11], who proposed the use default logic to model authorization and control rules. Default logic is a very expressive framework. However, the approach suffers from several drawbacks. First, default rules may not be conclusive – given an access request posed by a user, they may neither authorize nor deny the request. Second, the language of default logic is not even semi-decidable. Third, no conflict resolution mechanisms are articulated in their framework. In contrast, we have developed a compact language that is adequate to express all authorization policies, but is still computable in polynomial time. Additionally, in our framework, user's access requests are handled conclusively – they are either authorized or denied.

The work presented in this paper extends a previous proposal by us [7]. Major extensions concerns the consideration of authorization for roles and the investigation of how different security policies can be represented in our framework.

The remainder of this paper is organized as follows. Section 2 introduces the notations and definitions that will be used in the paper. Section 3 describes the logic language through which authorizations and rules can be specified. Section 4 defines authorization specifications and their correctness. Section 5 discusses possible approaches to propagation and conflict resolution and shows how each of this can be represented in our language. Section 6 illustrates how different constraints that can be required on the authorizations or on the system working can be included as rules in the specifications. Finally, Section 7 concludes the paper.

2 Basic assumptions, notations and definitions

In this section we introduce the basic elements on which our model is based and introduce notations and definitions that will be used in the rest of this paper.

2.1 Objects

We assume that there is a set Obj of objects defined in the system on which some *actions* can be executed. The choice of the specific objects and actions depends on the system to which the model is applied. For instance, in a file system objects will be the files and directories stored and the possible actions will be `read`, `write`, and `execute`. In a relational database system, the database, tables, tuples or columns within a table, and application programs will be considered as objects whereas actions will be represented by the different modes through which objects can be accessed such as `select`, `insert`, `update`, and `run` (the latter being applicable only to application programs).

To allow the specification of authorizations on sets of objects, we allow objects to be grouped into *types*. Again the types T to be considered are specific to the system to which the authorization model is applied and are not included in the authorization model itself. Types can correspond to the elements of the underlying data model. For example, in an operating system environment types such as `file`, `directory`, `executable program`, can be defined. Types can also be defined with respect to the data contained in the objects or to the application/activity in which they are used. For instance, types `ps-files`, `tex-files`, `dvi-files`, can be defined grouping together the postscript, latex, and dvi files respectively.

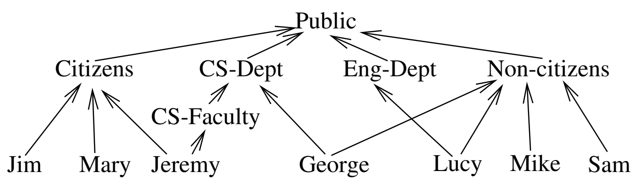

Figure 1. An example group hierarchy

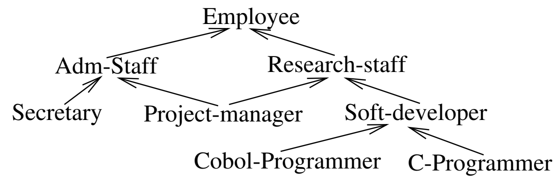

Figure 2. An example role hierarchy

2.2 Authorization's subjects

We consider three different kinds of subjects to which authorizations can be granted. These are *users* U, *groups* G, and *roles* R. Users are individuals connecting to the system and allowed to submit requests. Groups are sets of users; groups can be nested. Roles are *named* collection of privileges needed to perform specific activities in the system.

Let us first examine groups. A group is defined as a set composed of individual users or other groups. Groups do not need to be disjoint; a user or a group can belong to more than one group. We say that the membership of a subject s in a group G is *direct* if s is defined as a member of G. It is *indirect* if either $s = G$ or there exists a sequence $s_1, \ldots, s_n, n > 1$ such that s_i is a direct member of s_{i+1}, for $i = 1, \ldots, n-1$. The only constraint on the groups is that the membership relationship be acyclic, i.e., if a subject s_i is a member (directly or indirectly) of another subject s_j, with $s_i \neq s_j$, then s_j cannot be a member of s_i. The group membership relationship can be represented as a graph where nodes are users and groups and an arc directed from s_i to s_j indicates that s_i is a direct member of s_j. Users are the leaves of this graph. We refer to this graph as *group-hierarchy*. An example group-hierarchy is illustrated in Figure 1.

Throughout the rest of this paper, we will deal with group-hierarchical data systems. A data system with no hierarchy is actually captured as a data system with a hierarchy where nodes are users and there are no arcs between users. The only membership relationship holding in such a hierarchy is the indirect membership of a user to itself.

Roles are named collection of privileges. Intuitively, a role identifies a task that users need to execute to perform organizational activities. Example of roles can be `secretary`, `dept-chair`, `programmer`, `payroll-officer`, and so on. To enable users to successfully execute the task, each role has some authorizations associated with it (see Section 3). By assuming a given role, a user is allowed to perform the actions for which the role is authorized.

Like groups, roles can also be organized in a hierarchy. A role is a `sub-role` of another role if it is a specialization of it. For instance, `Fortran-programmer` and `Cobol-programmer` can be both seen as a specialization of `programmer`. The role hierarchy can also be represented by a graph where nodes are the different roles and a directed arc from role r_i to role r_j indicates that r_i is a specialization of r_j. The concepts of direct and indirect membership defined for groups apply to roles as well. Figure 2 illustrates an example role hierarchy.

In view of the different semantics groups and roles carry, we require that the group and role hierarchies to be disjoint (i.e., same subject cannot appear in both hierarchies). Note however that a same "concept" can be seen both as a group and as a role. To understand the difference between groups and roles, consider the following example. We could define a group, called `G_programmer`, consisting all users who are programmers. Any authorizations specified for `G_programmer` are propagated to its members. Thus, if an authorization to read `tech-reports` is given to `G_programmer`, its members can exercise this right. We could also define a role, called `R_programmer`, and associate to it those privileges that are related to the programming activity and necessary for the programmers to perform their jobs (such as compiling, debugging, writing reports, and so on). These privileges can be exercised by authorized users only when they choose to assume the role `R_programmer`.

2.2.1 Differences between groups and roles

A major difference between groups and roles is that roles can be "activated" and "deactivated" by users at their discretion while group membership always applies. In this section we clarify how role activation and deactivation changes the privileges of users.

We refer to all roles that a user is allowed to assume as *potential* roles. When a user activates a role, the role becomes *active* for the user. To take active roles into consideration when processing access requests, we introduce the concept of a *requestor subject* (i.e., a subject who requests access to objects). A requestor subject is a pair whose first element is a user (the user who enters the command) and whose second element is the set of roles currently active for the user (the set is empty if the user does not have any active roles).

If no roles are active for a user, the user will be able to exercise all the privileges s/he has either as an individual (i.e., explicitly granted to him/her) or as a member of groups to which s/he belongs. Remember that group membership always applies (i.e., users *cannot* activate or deactivate group memberships at will). By activating a role, a user will be allowed to exercise the privileges for which the role is authorized. A user can assume more than one role at a time. In this case s/he will be able to exercise the union of the privileges of all these roles.[1] How this union is to be determined and whether the user looses his/her personal privileges upon activation of some roles depend on the access decision policy to be applied. An interesting aspect of our approach is that our general language can be used to express and enforce different policies. Users can choose, when stating the authorization specifications, the access control policy to be applied. We will discuss some possible policies in Section 5. Moreover, in Section 6 we will illustrate how different kinds of constraints that can be imposed on the roles users can be authorized to activate or that can activate simultaneously [9] can be easily represented as rules in our language.

Note that the fact that the authorizations given to a role are applicable only when that role is active for a user has two advantages. First, it gives the user all those privileges that are needed to perform a task. Second, it is consistent with the "principle of least privilege": each process is *confined* to those actions needed to perform the task. This acts as a defense against malicious attacks that may aim to exploit the role's authorizations.

3 The Authorization Specification Language (ASL)

In this section, we introduce the basic constructs of our authorization language. We will show in subsequent sections how these constructs can be used by the System Security Officer (SSO) to specify authorizations and rules.

We start by defining an authorization policy.

Definition 3.1 (Authorization Policy) *An* authorization policy *is a mapping that maps 4-tuples* (o, u, R, a) *consisting of an object, a user, a role set, and an action, respectively to the set* $\{\texttt{authorized}, \texttt{denied}\}$.

Given a set of actions A, we define a set of signed authorization types SA as $\{+a, -a \mid a \in \mathsf{A}\}$. **ASL** is a logical language created from the following alphabet:

1. **Constant Symbols:** Every member of $\mathsf{Obj} \cup \mathsf{T} \cup \mathsf{U} \cup \mathsf{G} \cup \mathsf{R} \cup \mathsf{A} \cup \mathsf{SA} \cup \mathbb{N}$ where $\mathbb{N}$ denotes the set of natural numbers. Remember 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 authorizations respectively.

2. **Variable Symbols:** There are eight sets V_o, V_t, V_u, V_g, V_r, V_R,, V_a, V_{sa} of variable symbols ranging over the sets Obj, T, U, G, R, 2^{R}, A, and SA, respectively.

   In the following, we refer to elements of a set X and variables ranging over the set as "X terms". For instance, variables in V_o and members of Obj are object terms. We collectively refer to user, role, and group terms as subject terms.

3. **Predicate Symbols:** The following predicate symbols are considered.

   (a) A ternary predicate symbol, `cando`. The first argument of `cando` is an object term, the second is a subject term, and the third is a signed authorization term. The predicate `cando` represents authorizations explicitly inserted by the SSO.

   (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.

   (c) A ternary predicate symbol, `do`, with the same arguments as `cando`. The predicate `do` represents the authorizations that hold for each subject on each object. It enforces the conflict resolution policy.

   (d) A 4-term predicate symbol, `grant`. The first argument is an object term, the second argument is a user term, the third argument is a role-set term, and the fourth argument is a signed action term. The predicate `grant` represents the accesses to be allowed or denied to each requestor subject on each object. It enforces the access control policy.

   (e) A 5-term predicate symbol, `done`. The first argument is an object term, the second argument is a user term, the third argument is a role-set term, the fourth argument is an unsigned action term, and the fifth argument is a natural number. Done rules represent the accesses executed by requestor subjects.

   (f) A binary predicate `active` whose first argument is a user term and second argument is a role term. It captures the concept of active role/s for a user.

   (g) Two binary predicates `dirin` and `in` that take as arguments two subjects s_1, s_2. They capture the direct and indirect membership relationship between subjects.

   (h) A binary predicate `typeof` that takes as arguments, an object o and an object type t. It captures the grouping relationship between objects.

[1] Precisely speaking this is not actually a union since negative privileges also need to be considered.

(i) A predicate symbol `error`, with no arguments. If $\texttt{error}()$ 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.

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 will use the expression *literal* to denote an atom or its negation. For instance, if OT, ST, and SAT are object terms, subject terms, and signed action terms respectively then $\texttt{cando}(OT, ST, SAT)$ and $\neg\texttt{cando}(OT, ST, SAT)$ are examples of literals.

We now define the rules that can be expressed in our language.

Definition 3.2 (Done Rule) *A* done rule *is a rule of the form:*

$$\texttt{done}(o, s, R, a, t) \quad \leftarrow \quad .$$

where o, s, R, a and t are elements of $\mathsf{Obj}, \mathsf{U}, 2^{\mathsf{R}}, \mathsf{A}$ *and* $\mathbb{N}$ *respectively. Note that "done" rules are facts, as their body is always empty.*

Done rules are specified only by the system upon execution of accesses. If $\texttt{done}(\texttt{o}, \texttt{u}, \texttt{R}, \texttt{a}, \texttt{t})$ is true, then user `u` with roles in `R` active has executed action `a` on object `o` at time `t`. Done rules are useful for implementing those policies in which future accesses of a user are based on the accesses that the user has exercised in the past (as in the case of the Chinese Wall policy [5]).

Definition 3.3 (Authorization Rule) *An* authorization rule *is a rule of the form:*

$$\texttt{cando}(o, s, < sign > a) \quad \leftarrow \quad L_1 \& \ \ldots \& \, L_n .$$

where o, s, and a are elements of $\mathsf{Obj}, \mathsf{U} \cup \mathsf{G} \cup \mathsf{R}$, *and* A *respectively, $n \geq 0$, $< sign >$ is either $+$ or $-$ and, for each $0 < i \leq n$, L_i, either a* `in` *a* `dirin`*, or* `typeof` *literal.*

Authorization rules are specified by the SSO to allow or deny accesses to subjects. The literals in the right hand side of the rules are used to specify conditions that must be verified for the authorization to hold.

Example 3.1 *Consider the following authorization rules:*

$$\begin{array}{rcl} \texttt{cando}(\texttt{file1}, \texttt{CS-Dept}, +\texttt{write}) & \leftarrow & . \\ \texttt{cando}(\texttt{file2}, s, +\texttt{write}) & \leftarrow & \texttt{in}(s, \texttt{CS-Faculty}). \\ \texttt{cando}(o, \texttt{Secretary}, +\texttt{write}) & \leftarrow & \texttt{typeof}(o, \texttt{Letters}). \\ \texttt{cando}(\texttt{file1}, \texttt{George}, -\texttt{read}) & \leftarrow & . \end{array}$$

The first rule states that group `CS-Dept` *is authorized to write file* `file1`*. The second rule states that all members of group* `CS-Faculty` *can write* `file2`*. The third rule states that role* `Secretary` *can write objects of type* `Letters`*. Finally, the fourth rule states that* `George` *cannot read* `file1`.

□

The reader may wonder why the model allows `in` and `dirin` statements in the body of rules when authorizations can be specified for groups. The reason for this is that 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 a subject and the authorizations the subject may inherit from their super-subjects. Specifying an authorization for a group is therefore different, in general, from specifying an authorization for all subjects belonging to the group. To illustrate consider the first two rules of Example 3.1. The first rule specifies an authorization for group `CS-Dept`. The 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 by the SSO. By contrast the second rule specifies an authorization for the members of `CS-Faculty`. This latter authorization is is therefore a short hand that can be used instead of specifying a different authorization for each single member of the group.

The SSO states authorizations through `cando` rules. From the authorizations so specified, further authorizations can be derived by the system through the application of specified derivation rules. To distinguish authorizations explicitly stated by the SSO from the authorizations derived by the system through derivation rules, we use the predicate `dercando` for derived authorizations. Derivation rules are formally defined as follows:

Definition 3.4 (Derivation Rule) *A* derivation rule *is a rule of the form:*

$$\texttt{dercando}(o, s, < sign > a) \quad \leftarrow \quad L_1 \& \ \ldots \& \, L_n .$$

where o, s, and a are elements of $\mathsf{Obj}, \mathsf{U} \cup \mathsf{G} \cup \mathsf{R}$, *and* A *respectively, $< sign >$ is either $+$ or $-$, $n \geq 0$, and $L_1, \ldots, L_n$ are either* `cando`, `dercando`, `done`, `in`, `dirin`, *or* `typeof` *literals. All* `dercando`*-literals appearing in the body of a derivation rule must be positive.*

Example 3.2 *The derivation rules*

$$\begin{array}{rcl} \texttt{dercando}(o, s, +a) & \leftarrow & \texttt{cando}(o, s', +a) \& \ \texttt{in}(s, s'). \\ \texttt{dercando}(o, s, -a) & \leftarrow & \texttt{cando}(o, s', -a) \& \ \texttt{in}(s, s'). \end{array}$$

enforce implied authorizations from subjects to the subjects below them in the hierarchies (i.e., from a group to its members and from a role to its specialized roles). □

Beside being used for expressing propagation of authorizations along subject's hierarchies, due to their generality, derivation rules can be used to express different kinds of implication relationships between authorizations. For instance, the derivation of an authorization on the basis of the presence or the absence of another authorization, as proposed in [2], can be enforced by putting the authorization to be derived in the left hand side of the rule and the condition for the derivation in the right hand side of the derivation rule. By using variables instead of ground terms and by combining different literals, we can also express a large variety of implication relationships.

Example 3.3 *Consider the following derivation rules:*

$$\begin{aligned}\texttt{dercando}(\texttt{file1}, s, -\texttt{read}) &\leftarrow \texttt{dercando}(\texttt{file2}, s', \texttt{read})\\ &\quad \& \ \texttt{in}(s, s")\& \ \texttt{in}(s', s").\\ \texttt{dercando}(o, s, -\texttt{write}) &\leftarrow \texttt{done}(o', s, \texttt{read})\\ &\quad \& \ \texttt{typeof}(o, \texttt{Exams})\\ &\quad \& \ \texttt{typeof}(o', \texttt{Solutions}).\end{aligned}$$

The first rule derives a negative authorization for a subject s to read `file1` *if there exist another subject s' and a group s'' such that s and s' both belong to s'' and s' is authorized to read* `file2`*.*
The second rule derives the negative authorization for a user to write an object of type `Exams` *if the user has read an object of type* `Solutions`*.* □

Derivation rules allow the derivation of authorizations on the basis of other authorizations, either derived or explicitly specified by the SSO.

`cando` and `dercando` rules may admit the derivation of both positive and negative authorizations for a given object, subject, and action. The concept of a *resolution rule*, given below, forces a decision to be made. Resolution rules can also be used to force a decision in case `cando` and `dercando` rules do not imply either a positive or a negative authorization for a given object, subject, and action. In such a case the decision is called the *default* decision.

Definition 3.5 (Resolution Rule) *A* resolution rule *is a rule of the form*

$$\texttt{do}(o, s, < sign > a) \quad \leftarrow \quad L_1 \& \ \ldots \& \ L_n.$$

where $o, s,$ and a are elements of Obj, U ∪ G ∪ R, *and* A *respectively, $< sign >$ is either $+$ or $-$, $n \geq 0$, and $L_1, \ldots, L_n$ are* `cando`, `dercando`, `in`, `dirin`, `done`, *or* `typeof` *literals and every variable that appears in any of the L_i's also appears in the head of this rule.*

A resolution rule states that a subject must be allowed/forbidden to exercise an authorization type on an object.

Note the difference between `do` rules on the one hand and `cando` and `dercando` rules on the other. `cando` and `dercando` rules are authorizations, either positive or negative, specified by the SSO or derived. These authorizations may conflict and, therefore, may not be obeyed by the system. *In contrast,* `do` *rules state what authorizations the system must consider valid for each authorization subject, on the basis of the existing authorizations, specified or derived.*

Example 3.4 *Consider the following rules:*

$$\begin{aligned}\texttt{do}(\texttt{file1}, s, +a) &\leftarrow \texttt{dercando}(\texttt{file1}, s, +a).\\ \texttt{do}(\texttt{file2}, s, +a) &\leftarrow \neg\texttt{dercando}(\texttt{file2}, s, -a).\end{aligned}$$

The first rule states that a subject can exercise an access on object `file1` *if s/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` *only if s/he does not have a negative authorization for it (i.e., the open policy is enforced on* `file2`*).* □

Authorization, derivation, and resolution rules establish whether a positive, a negative, or no authorization must hold for each subject and access. These rules by themselves are not sufficient to regulate access control. The reason for this is that, as we have already illustrated in Section 2, requestor subjects are composition of users and, possibly, roles. Hence, requestor subjects do not correspond with authorizations subjects to which resolution rules refer. Access control rules determine whether a requestor subject (i.e., a user with, possibly, some active roles), must be allowed or denied for an access depending on the authorizations specified for the user and/or for the roles. Access control rules are defined as follows.

Definition 3.6 (Access Control Rule) *An* access control rule *is a rule of the form:*

$$\texttt{grant}(o, u, rs, < sign > a) \quad \leftarrow \quad L_1 \& \ \ldots \& \ L_n.$$

where $L_1, \ldots, L_n$ are `cando`, `dercando`, `done`, `do`, `in`, `dirin`, *or* `typeof` *literals.*

If $\texttt{grant}(\texttt{o}, \texttt{u}, \texttt{R}, +\texttt{a})$ is true, then user `u` with active roles `R`, will be allowed to perform action `a` on object `o`. Similarly, if $\texttt{grant}(\texttt{o}, \texttt{u}, \texttt{R}, -\texttt{a})$ is true, then user `u` with active roles `R` will not be allowed to perform action `a` on object `o`.

Most of the time, the user and the role-set terms in access control rules will be variables. This is due to the fact that *access control rules* are typically used to specify the rules *generally applicable* to determine the access decisions (whether to allow or not allow an access on the basis of authorizations specified). They are not used to specify authorizations.

Example 3.5 *Consider the following rules:*

$$\begin{array}{lcl} \mathtt{grant}(o,u,R,+a) & \leftarrow & \mathtt{do}(o,s,+a)\ \&\ R=\emptyset. \\ \mathtt{grant}(o,u,R,+a) & \leftarrow & \mathtt{do}(o,r,+a)\ \&\ \neg\mathtt{do}(o,r,-a) \\ & & \&\ r\in R\ \&\ r'\in R. \end{array}$$

The first rule states that a request submitted by a user operating as a single individual (no current active role) is to be allowed if the user has a positive authorization for it.

The second rule states that a request by a user with some roles R active is to be allowed if at least one of the active roles is authorized for it and none of them is denied for it. □

Authorization, derivation, resolution, and access control rules are all we need to specify authorizations and access control decisions. Our language supports an additional ind of rule, called *integrity* rule, by which the SSO can define constraints that must hold on the specifications. Integrity rules can be specified on any of the literals or their combination. They are formally defined as follows.

Definition 3.7 (Integrity Rule) *An* integrity rule *is a rule of the form:*

$$\mathtt{error}() \quad \leftarrow \quad L_1\&\ \ldots\&\ L_n.$$

where $L_1,\ldots,L_n$ *are* `grant`, `cando`, `dercando`, `done`, `do`, `in`, `dirin`, `typeof`, *literals.*

The above rule derives an error every time the conditions in the right hand side of the rules are satisfied. Whenever a new rule r is to be inserted, the error rules are evaluated and the r accepted only if its insertion does not generated any error.[2] We note that integrity rules may be general or may be specific to an application. General rules control inconsistencies such as "no subject can be both authorized and denied for the same access". Application-dependent rules control inconsistencies appropriate for the application, such as "a subject cannot be authorized to read both `fileA` and `fileB`". Section 6 contains some examples illustrating the expressive power of integrity rules.

4 Authorization specifications

The authorization language we have presented in the previous section allows us to state authorization specifications.

[2] Note that in case r is a `done` rule, the fact that it cannot be inserted means that the execution of the corresponding access cannot be allowed.

An authorization specification is a collection of rules whose evaluation determines, for each access request that can be submitted, whether the requested access must be granted or denied. Authorization specifications are formally defined as follows.

Definition 4.1 (Authorization Specification) *An* authorization specification $\mathbf{AS}$ *consists of a set of authorization (*`cando`*), derivation (*`dercando`*), resolution (*`do`*), access control (*`grant`*), and integrity (*`error`*) rules.*

Every authorization specification in our language is a *stratified* datalog program. Upon each access request by user u with active roles R to execute action a on object o the program is evaluated and access allowed if and only if $\mathtt{grant}(o,u,R,+a)$ is true according to the semantics of the specifications. This access control checking can be performed in linear time w.r.t. the number of rules in $\mathbf{AS}$ [7].

An important aspect of authorization specifications is their correctness. An authorization specification is correct if it is both *consistent* and *complete*. By *consistent* we mean that for each possible access a *unique* access decision exists, i.e., the access is either to be granted or denied but not both. By *complete* we mean that for each access an access decision, either grant or deny, exists.

It is important to note that consistency and completeness apply to *access decisions* not to authorizations specified or derived. As a matter of fact, inconsistent authorizations or derivations can be allowed. Analogously, the presence of both positive and negative authorizations applicable to a subject can be allowed (consider for instance the case of a user activating two roles with contrasting authorizations). *Consistency* requires that the *access decision* be unique. The same consideration applies to completeness. Requiring access decision completeness does not imply requiring authorization specification completeness. Access decision completeness ensures that a decision can always be made. This can be true even if for some access no authorization is specified or derived (*default decision*).

Authorization specification correctness is defined as follows.

Definition 4.2 (Authorization Specification Consistency and Completeness) *An authorization specification is* complete *if for each 4-tuple* (o,s,R,a) *at least one of* $\mathtt{grant}(o,s,R,+a)$ *and* $\mathtt{grant}(o,s,R,-a)$ *is true. An authorization specification is* consistent *if for each 4-tuple* (o,s,R,a) $\mathtt{grant}(o,s,R,+a)$ *and* $\mathtt{grant}(o,s,R,-a)$ *cannot both be true.*

It is interesting to note how specification correctness can be stated and checked by using rules expressed in the ASL language. In particular, the following two integrity rules can be used to return an error if either completeness (first rule) or consistency (second rule) is not satisfied.

$$\texttt{error}() \leftarrow \neg\texttt{grant}(o, u, R, +a) \& \neg\texttt{grant}(o, u, R, -a).$$
$$\texttt{error}() \leftarrow \texttt{grant}(o, u, R, +a) \& \texttt{grant}(o, u, R, -a).$$

The correctness of authorization specifications can therefore be enforced by inserting the above rules in the specifications themselves. An alternative way of enforcing specification correctness is by imposing restrictions on the the rules that can be specified [7].

As we have stated completeness and consistency obviously apply to access decisions (it would not be proper to require them on the authorizations themselves). We note however that, due to the semantics of resolution rules, consistency (*not* completeness) of the do predicate must also be required. The consistency requirement for the conflict resolution decision is similar to that for access control decision. Resolution consistency requires that for each subject, object, and action, $\texttt{do}(o, s, +a)$ and $\texttt{do}(o, s, -a)$ cannot be both satisfied.

5 Derivation of authorizations and conflict resolutions along subject hierarchies

In our model, we consider two different and disjoint hierarchies: the group hierarchy and the role hierarchy. However, different models make different choices with respect to whether or how authorizations propagate along the hierarchy and the way conflicts between authorizations independently propagated to a subject are resolved. In this section, we will examine some of these different approaches and show how each of them can be easily represented with the use of few rules in our language. Recall that the ultimate goal of a security policy is to decide whether an authorization (positive or negative) holds for a given subject when the authorizations specified for the subject and for other subjects appearing in the hierarchies are taken into account. Two different aspects of the policy can be distinguished:

Derivation It regulates the propagation of authorizations along the hierarchy.

Conflict resolution It determines which authorization should take precedence when conflicting authorizations (i.e., different signs but identical actions and objects) exists for the *same* subject.

Below, we discuss different choices that have been made for each of these two aspects. Our discussion is in terms of a generic subject hierarchy, and applies to both group and role hierarchies. Our clarifying examples refer to the subject hierarchy shown in Figure 3. For the sake of simplicity, in the figures, we indicate the positive and/or negative authorizations held by a subject by writing the sign near the subject and omit the specification of the object and action. All authorizations in a figure are assumed to refer to the same object and action.

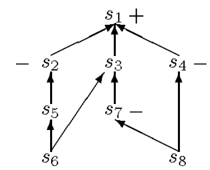

Figure 3. An example of subject hierarchy

5.1 Derivation of authorizations along subjects hierarchies

The derivation policy determines whether and how authorizations propagate from a subject to other subjects that are connected to it in a hierarchy. The first choice that must be made in this respect is whether the authorizations of a subject should propagate at all to other subjects. As for groups, the answer is obviously yes. Since groups do not appear in access requests (only users and roles do), the only way authorizations of groups can be effective is by propagating them to the user members of the groups.

The answer is not so obvious for roles. In most models that have roles, authorizations specified for a role are propagated to other roles specialized from it. Hence, each role inherits the authorizations of the roles of which it is a specialization. However, a different, more conservative, approach could be to consider the authorizations necessary for each role separately and do not propagate them.[3]

When the authorizations of a subject propagate along the hierarchy, the derivation policy must determine which authorizations propagate to a subject s in case s is a child-subject of subjects with contrasting authorizations. When the subjects for which the contrasting authorizations are specified are not connected by any path in the hierarchy (e.g., s_4 and s_7 in Figure 3), the contrasting authorizations must all be propagated to s leaving the decision of which authorization should win over the other to the conflict resolution policy. In contrast, when the subjects for which the contrasting authorizations are specified are related in the hierarchy (e.g. s_1 and s_2 in Figure 3), overriding policies can be applied. Three different approaches can be taken in this case, which we list below.

- *No overriding* All the authorizations are propagated (regardless of the presence of other conflicting authorizations).

[3] We note that the two approaches are not mutually exclusive. A hybrid approach could be taken in which some authorizations are propagated while others are not.

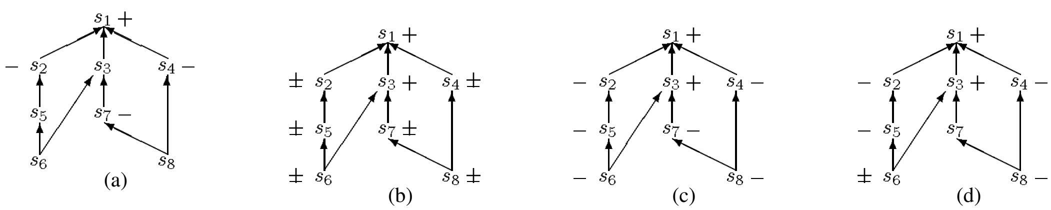

Figure 4. Authorizations derived from the specifications in Figure 3: (a) no propagation; (b) no overriding (c) sub-subject overriding; (d) path overriding

- *Sub-subject overrides* The authorization specified for a subject s overrides a conflicting authorization specified for a supersubject of s. In Figure 3, the negative authorization of s_2 overrides the authorization of s_1 for s_2. In this case, only the authorization of s_2, not the authorization of s_1, will be propagated to s_6.

- *Path overrides* The authorization specified for a subject of s overrides a conflicting authorization specified for a supersubject of s *only* along the paths passing from s. The overriding has not effect on other paths. In Figure 3, the negative authorization of s_2 overrides the authorization of s_1 for s_2 and all its members only for the membership paths passing from s_2. The authorizations of s_1 could still still propagate to the subjects below it in the hierarchy through other membership paths. In particular, the authorization specified for s_1 propagates to s_6 through the membership path from s_6 to s_1 passing from s_3.

Figure 4 illustrates the authorizations derived from the specification of Figure 3 in each of the approaches above. The ASL rules implementing each of the approaches are illustrated in Figure 5.

5.2 Conflict resolution

In case a subject holds both a positive and a negative authorization for the same access (e.g., subject s_6 in Figure 4(b,d)), the conflict resolution policy determines which of the authorization should be enforced. Four different approaches could be taken:

- *No conflict* It requires that no conflict arises. The presence of a conflict is considered an error.

- *Denials take precedence* The negative authorization is enforced over the positive one (subject s_6 will be considered denied).

- *Permissions take precedence* The positive authorization is enforced over the negative one (subject s_6 will be considered authorized).

- *Nothing takes precedence* None of the two authorizations can be considered as prevailing over the other. (The positive and the negative authorizations nullify each other.) The final result is equivalent to the case where no authorization had actually been specified (no decision will be taken for subject s_6).[4]

Figure 6 shows the ASL rules implementing the different approaches to conflict resolution.

6 Expressiveness of the ASL

From the previous section, it should already be clear how our language can be used to represent many of the models present in the literature. In particular, most models use some of the different approaches we have discussed in the previous section and, hence, their mapping is almost immediate.

The classical *closed* (*open*) policies can be easily represented by allowing the specification of only positive (negative) authorizations and by appropriate access control rules. The restriction that only positive (negative) authorizations can be specified, can be enforced through an integrity rule. As for access control, the rules enforcing the closed (open) policy will derive a positive (negative) grant if a positive (negative) authorization exists and a negative (positive) grant otherwise. Examples of rules enforcing the closed and open policies have been given in Example 3.4.

To illustrate the expressiveness of our language, we now consider several policies that do not fall in any of the categories given in the previous section and show how our language can be used to represent these policies .

6.1 Representing Authorizations Models

We show how the SeaView [8] authorization model can be represented using our language. In SeaView, authorization subjects can be either users or groups. Groups are

[4] Note that this does not mean that no access decision can be taken. Access decisions are taken on the basis of the access control rules, not on the basis of the resolution rules. The completeness of the access control rules ensures that an access decision will always be taken even if no authorization decision can be taken by the resolution 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	$\texttt{dercando}(o, s, +a)$	$\leftarrow$	$\texttt{cando}(o, s', +a) \& \ \texttt{in}(s, s').$
	$\texttt{dercando}(o, s, -a)$	$\leftarrow$	$\texttt{cando}(o, s', -a) \& \ \texttt{in}(s, s').$
Sub-subject overrides	$\texttt{dercando}(o, s, +a)$	$\leftarrow$	$\texttt{cando}(o, s', +a) \& \ \neg\texttt{cando}(o, s'', -a) \& \ \texttt{in}(s, s') \& \ \texttt{in}(s, s'') \& \ \texttt{in}(s'', s') \& \ s'' \neq s'.$
	$\texttt{dercando}(o, s, -a)$	$\leftarrow$	$\texttt{cando}(o, s', -a) \& \ \neg\texttt{cando}(o, s'', +a) \& \ \texttt{in}(s, s') \& \ \texttt{in}(s, s'') \& \ \texttt{in}(s'', s') \& \ s'' \neq s'.$
Path overrides	$\texttt{dercando}(o, s, +a)$	$\leftarrow$	$\texttt{cando}(o, s, +a).$
	$\texttt{dercando}(o, s, -a)$	$\leftarrow$	$\texttt{cando}(o, s, -a).$
	$\texttt{dercando}(o, s, +a)$	$\leftarrow$	$\texttt{dercando}(o, s', +a) \& \ \neg\texttt{cando}(o, s, -a) \& \ \texttt{dirin}(s, s').$
	$\texttt{dercando}(o, s, -a)$	$\leftarrow$	$\texttt{dercando}(o, s', -a) \& \ \neg\texttt{cando}(o, s, +a) \& \ \texttt{dirin}(s, s').$

Figure 5. Different approaches to the derivation of authorizations

No conflict	$\texttt{do}(o, u, +a)$	$\leftarrow$	$\texttt{dercando}(o, u, +a).$
	$\texttt{do}(o, u, -a)$	$\leftarrow$	$\texttt{dercando}(o, u, -a).$
	$\texttt{error}()$	$\leftarrow$	$\texttt{dercando}(o, u, +a) \& \ \texttt{dercando}(o, u, -a).$
Denials take precedence	$\texttt{do}(o, u, +a)$	$\leftarrow$	$\texttt{dercando}(o, u, +a) \& \ \neg\texttt{dercando}(o, u, -a).$
	$\texttt{do}(o, u, -a)$	$\leftarrow$	$\texttt{dercando}(o, u, -a).$
Permission takes precedence	$\texttt{do}(o, u, +a)$	$\leftarrow$	$\texttt{dercando}(o, u, +a).$
	$\texttt{do}(o, u, -a)$	$\leftarrow$	$\texttt{dercando}(o, u, -a) \& \ \neg\texttt{dercando}(o, u, +a).$
Nothing takes precedence	$\texttt{do}(o, u, +a)$	$\leftarrow$	$\texttt{dercando}(o, u, +a) \& \ \neg\texttt{dercando}(o, u, -a).$
	$\texttt{do}(o, u, -a)$	$\leftarrow$	$\texttt{dercando}(o, u, -a) \& \ \neg\texttt{dercando}(o, u, +a).$

Figure 6. Different approaches to conflict resolutions

sets of individual users and *cannot* be nested. Negative authorizations are not considered. A special privilege `null`, meaning no access allowed, is used instead. A user can exercise the privileges of only one group at a time (SeaView groups behave somewhat like our roles.) A requestor subject is the pair consisting of the user and the group the user has currently activated. The request of a subject to exercise a given action on an object is granted only if any of the following conditions hold: (*i*) the user has the authorization for the access and does not have a null authorization for it, or (*ii*) the user does not have any authorization at all (for any action) on the object and the group has the authorization for the access and does not have a null authorization on the object.

Due to the way groups are used in SeaView, they can be represented as roles in our model. Thus, SeaView can be expressed using the following ASL rules:

$$\begin{aligned}
\texttt{do}(o, u, s, +a) &\leftarrow \texttt{cando}(o, u, +a) \& \ \neg\texttt{cando}(o, s, +\texttt{null}). \\
\texttt{grant}(o, u, R, +a) &\leftarrow \texttt{do}(o, u, +a). \\
\texttt{grant}(o, u, R, +a) &\leftarrow \neg\texttt{cando}(o, u, +a') \& \ \texttt{active}(u, r) \\
&\qquad \& \ \texttt{do}(o, r, +a). \\
\texttt{grant}(o, u, R, -a) &\leftarrow \neg\texttt{grant}(o, u, R, +a). \\
\texttt{error}() &\leftarrow \texttt{in}(s, s') \& \ \neg\texttt{in}(s, s') \& \ s \neq s'. \\
\texttt{error}() &\leftarrow \texttt{do}(o, u, s, -a). \\
\texttt{error}() &\leftarrow \texttt{active}(u, r) \& \ \texttt{active}(u, r') \& \ r \neq r'.
\end{aligned}$$

The resolution rule states that an authorization for an action on an object specified for a user/group is valid if the user/group does not have a null authorization on the object. The three grant rules enforce access control. The two positive rules determine when the access is to be allowed and enforce conditions (*i*) and (*ii*) above respectively. The negative rule states that an access request for which no positive grant can be derived is to be denied. This rule provides completeness of the specifications. The three error rules enforce the different constraints of the model, that is: groups cannot be nested (first rule), no negative authorizations can be specified (second rule), and a user can activate at most one role at a time (third rule).

6.2 Representing constraints

In this section, we illustrate the power of the integrity rules by showing how various constraints can be easily represented as integrity rules in ASL specifications.

Incompatible groups Two groups are said to be *incompatible* if they cannot have common members (i.e., no subject can belong to both groups). For instance, groups `Non-citizens` and `Citizens` are incompatible. Group incompatibility can be easily represented by an error rule whose body contains the `in` predicates of the incompatible groups, as follows:

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

Incompatible roles assignment Two roles are said to be *incompatible* if they cannot be activated by the same user. As an example, consider the roles `participant` and `examiner` with the constraint that a user allowed to activate role participant cannot be allowed to activate (at the same or at a different time) the role examiner, and conversely. This constraint can be represented by an integrity rule that returns an error if a user is granted the permissions to activate both roles. More generally, the incompatibility of n roles r_1, ..., r_n can be expressed as follows:

$$\begin{aligned}\texttt{error}() \quad \leftarrow \quad & \texttt{grant}(r_1, u, R, \texttt{activate}) \& \ \ldots \\ & \& \ \texttt{grant}(r_n, u, R, \texttt{activate}).\end{aligned}$$

Incompatible roles activation The *activation* of n roles is incompatible if the roles cannot be all activated *simultaneously*. Note the difference between role activation incompatibility and role assignment incompatibility. In role activation incompatibility, a user cannot activate the incompatible roles *simultaneously*, but s/he can activate them at different times. Role activation incompatibility may be required to prevent a user to operate with the union of the privileges of the roles since this would give the user (more precisely, a process executing on his/her behalf) too much power, possibly allowing exploitation of the allowed accesses. Let r_1, ..., r_n be incompatible roles. The incompatible role activation constraint can be represented by an integrity rule that returns an error if, according to the specifications, a user has all the incompatible roles active:

$$\texttt{error}() \quad \leftarrow \quad \texttt{active}(u, r_1) \& \ \ldots \& \ \texttt{active}(u, r_n).$$

Static separation of duty Static separation of duty refers to the fact that a certain set of accesses cannot be allowed for the same subject. The reason for this is that the union of the accesses would give the subject too much power. For instance, consider the operations of submitting, evaluating, and approving the budget. A static separation of duty requirement indicates that a same subject cannot be authorized for all the three operations above. This requirement can be referred to either authorization subjects or requestor subjects. The constraint that a same authorization subject cannot be authorized for all the three operations above can be expressed as follows:

$$\begin{aligned}\texttt{error}() \quad \leftarrow \quad & \texttt{do}(\texttt{budget}, s, \texttt{submitting}) \\ & \& \ \texttt{do}(\texttt{budget}, s, \texttt{evaluating}) \\ & \& \ \texttt{do}(\texttt{budget}, s, \texttt{approving}).\end{aligned}$$

The rule above regulates the authorizations of each authorization subject individually taken. A user could however be able to execute all the actions above by employing, either simultaneously or at a different time, the roles which are authorized for the different actions. To avoid this situation, it is sufficient to express the separation of duty constraint with reference to requestor subjects as follows:

$$\begin{aligned}\texttt{error}() \quad \leftarrow \quad & \texttt{grant}(\texttt{budget}, u, R, \texttt{submitting}) \\ & \& \ \texttt{grant}(\texttt{budget}, u, R', \texttt{evaluating}) \\ & \& \ \texttt{grant}(\texttt{budget}, u, R'', \texttt{approving}).\end{aligned}$$

The rule states that a same user, even if activating different roles, cannot be allowed for the execution of all the actions above.

Dynamic separation of duty The separation of duty constraints above refers to authorizations and accesses allowed. In other words they constraint the authorization specifications. There are cases where separation of duty constraints are not to be imposed on the authorizations but on their use. In this case, a user can potentially execute any operation in the set. However s/he cannot execute all of them. By executing some s/he will automatically rule out the possibility of executing the others. Note the difference between this kind of separation of duty constraint, which we refer to as *dynamic*, and the one in the previous example, which we refer to as *static*. In static separation of duty the SSO must specify the authorizations in such a way that no subject will ever be granted all the actions in the set. Hence, which actions a subject will, or will not, be allowed to execute, is determined by the SSO. In dynamic separation of duty, which actions the user executes is determined by the user. To illustrate the usefulness of dynamic separation, consider an office with ten clerks. A group Clerk is defined to which these users belong and to which the authorizations for submitting, approving, and paying orders are given.[5] Separation of duty requires that no user must be able to execute all the three actions on a *same* order. Which of the actions a user execute is not predetermined. However, if a user executes two s/he must be forbidden to execute the other. This constraint can be expressed by the following rule.

$$\begin{aligned}\texttt{error}() \quad \leftarrow \quad & \texttt{done}(o, u, R, \texttt{submitting}, t) \\ & \& \ \texttt{done}(o, u, R', \texttt{approving}, t') \\ & \& \ \texttt{done}(o, u, R'', \texttt{approving}, t'') \\ & \& \ \texttt{typeof}(o, \texttt{Order}).\end{aligned}$$

Chinese Wall The Chinese Wall constraint [5] can be seen as a special kind of dynamic separation of duty. In the Chinese Wall policy, objects are grouped into *company datasets*, e.g., Company-A and Company-B. Company datasets whose organizations are in competitions are then grouped together into *conflict of interest classes*. If a user accesses an object in a company dataset cd s/he cannot be allowed anymore to access any object in a company datasets that appear in a conflict of interest class with cd. For instance, if Company-A and Company-B are in a same conflict of interest class, a user who has accessed an object of Company-A will not be able to access any object in Company-B and vice versa. A possible way to represent this constraint in our model is by representing company datasets as types. An integrity rule can then be specified that returns an error if a user accesses objects

[5] Note that the example can be expressed also with the use of roles. In this case a role Clerk is considered that users can activate in order to execute the actions on orders.

of two datasets in the same conflict of interest class. For instance, the following rule enforces the constraint for the two datasets above:

$$\texttt{error}() \leftarrow \texttt{done}(o', u, R, a', t) \& \texttt{done}(o, u, R', a, t') \& \texttt{typeof}(o, \texttt{Company-A}) \& \texttt{typeof}(o', \texttt{Company-B}).$$

The ones reported above are only some examples of the constraints that can be represented in our language. Several other constraints can be imagined and a rule enforcing them easily found. Also, the general constraints represented above can be "personalized" or slightly modified to adapt to different requirements. For instance a more restrictive interpretation of the Chinese Wall policy can forbid the execution of the two actions above not only to a single user, but also to users belonging to a single group. In such a case, if a user accesses an object of type `Company-A` no user of his group/s will be allowed to access objects in `Company-B`. This can be expressed as follows:

$$\texttt{error}() \leftarrow \texttt{done}(o, u, R, a, t) \& \texttt{done}(o, u', R', a, t') \& \texttt{typeof}(o, \texttt{Company-A}) \& \texttt{typeof}(o', \texttt{Company-B}) \& \texttt{in}(u, G) \& \texttt{in}(u', G).$$

7 Conclusions

In this paper we have proposed a logical language for the specification of authorizations on which such a model can be based. The language allows users to specify, together with the authorizations, the policy according to which access control must be enforced. Different policies can be specified on different objects, according to the needs of the users. The language supports both the concept of groups and roles and allows the specification of different rules regulating the access control decisions. We have illustrated how security specifications are stated in our language and shown how different control policies can be represented. We have also stated consistency and completeness constraints that security specifications are required to obey. Moreover we have illustrated how different constraints that are generally required, but very seldom supported by the access control systems, can be represented in our language. 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.

Our paper leaves space 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 the consideration of administrative policies regulating the insertion of the different rules by the users. We also plan to investigate how our model can be applied in the representation and enforcement of complex organization's security policies, such as those of financial or health-care institutions.

References

[1] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of deductive databases*, pages 89–148. Morgan Kaufmann, San Mateo, 1988.

[2] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. A temporal access control mechanism for database systems. *IEEE Trans. on Knowledge and Data Engineering*, 8(1):67–80, February 1996.

[3] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 94–107, Oakland, CA, May 1996.

[4] M. Branstad, H. Tajalli, F. Mayer, and D. Dalva. Access mediation in a message passing kernel. In *Proc. IEEE Symp. on Security and Privacy*, pages 66–72, Oakland, CA, May 1989.

[5] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proc. Symp. on Security and Privacy*, pages 215–228, Oakland, CA, May 1989.

[6] T. Fine and S. E. Minear. Assuring distributed trusted mach. In *Proc. IEEE Symp. on Security and Privacy*, pages 206–218, Oakland, CA, May 1993.

[7] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD Conf. on Management of Data*, Tucson, AZ, May 1997.

[8] T. F. Lunt. Access control policies for database systems. In C. E. Landwehr, editor, *Database Security II: Status and Prospects*, pages 41–52. North-Holland, Amsterdam, 1989.

[9] R. Sandhu, E. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, pages 38–47, February 1996.

[10] O. S. Saydjari, S. J. Turner, D. E. Peele, J. F. Farrell, P. A. Loscocco, W. Kutz, and G. L. Bock. Synergy: A distributed, microkernel-based security architecture, version 1.0. Technical report, National Security Agency, Ft. George G. Meade, MD, November 1993.

[11] T. Y. C. Woo and S. S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.