A Unified Framework for Enforcing Multiple Access Control Policies*

Sushil Jajodia
Center for Secure Information Systems and
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030-4444, USA
jajodia@gmu.edu

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

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

Elisa Bertino
Dipartimento di Scienze dell'Informazione
Università di Milano, 20135 Milano, Italy
bertino@dsi.unimi.it

Abstract

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 be actually applied within a given system. However, protection requirements within a system can vary dramatically, and no single policy may simultaneously satisfy them all.

In this paper we present a flexible authorization manager (FAM) that can enforce multiple access control policies within a single, unified system. FAM is based on a language through which users can specify authorizations and access control policies to be applied in controlling execution of specific actions on given objects. We formally define the language and properties required to hold on the security specifications and prove that this language can express *all* security specifications. Furthermore, we show that all programs expressed in this language (called FAM/CAM-programs) are also guaranteed to be consistent (i.e., no conflicting access decisions occur) and CAM-programs are complete (i.e., every access is either authorized or denied). We then illustrate how several well-known protection policies proposed in the literature can be expressed in the FAM/CAM language and how users can customize the access control by specifying their own policies. The result is an access control mechanism which is flexible, since different access control policies can all coexist in the same data system, and extensible, since it can be augmented with any new policy a specific application or user may require.

*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.

1 Introduction

Several access control policies have been proposed in the literature for controlling access to information. Correspondingly, several authorization models have been formalized and access control mechanisms enforcing them implemented. Each model, and its corresponding enforcing 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 protection requirements that may arise over time. For instance, each of us deals with data protection in different ways. We may have information that we want to keep completely private, information we want to share with everybody, information we want to share with almost everybody (with a few exceptions), and information we want to share with only a select few. When a system *imposes* a specific policy on a user, that user has to work within the confines/limitations thus imposed. Specifying some protection requirements may become trivial, while others may become impossible.

Furthermore, with the advent of the information superhighway, a database administrator (DBA) or system security officer (SSO) may be responsible for providing secure, authorized access to a wide variety of distributed objects including some files, some relations, some object bases, some images, etc. Figure 1 shows a simple example of some objects that an SSO is providing access to. In this example, 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 (OO) data. Databases, operating systems, and file systems (and in general systems that create and manipulate objects) must provide a way for the SSO to apply different access control policies to different classes of data objects.

In this paper, we define a *Flexible Authorization Manager* (FAM) that consists of two parts: an *authorization language* within which an SSO may specify authorizations and authorization policies (perhaps using a GUI), and an *authorization request processor* that processes, at run-time, a user's access request against the SSO's specifications.

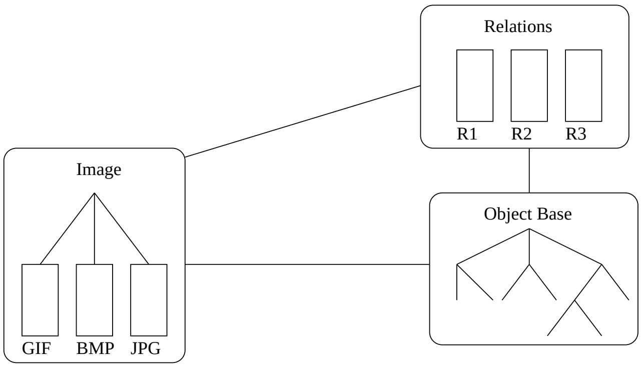

Figure 1: Example of Distributed, Heterogeneous Objects

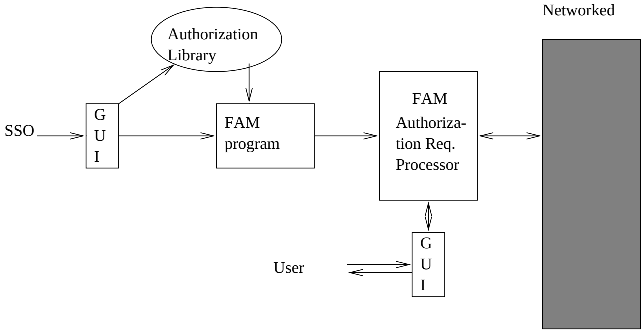

Figure 2: Architecture of the Flexible Authorization Manager (FAM)

The FAM Authorization Language is a very compact logical language using which an SSO may:

1. specify which users/user groups may access which objects,

2. specify which authorization policies apply (by selecting authorization policies stored in an Authorization Library) to one or more classes of objects, and

3. express/create new authorization policies in the language that may then be stored in the library.

The result of the above steps is a program called a FAM-program in our FAM Authorization language (a more "compact" version of FAM is called a "Compact Authorization Manager" and a CAM-program is also defined). Once the FAM-program has been created, our framework may broker user access to objects as follows: Whenever a user u asks for access to object o, our system will convert this access request to a query $Q_{u,o}$ that will be posed to the FAM-program (using the Authorization Request Processor) created by the SSO. If the FAM-program returns "yes" then the user is given the desired access, otherwise the user's request is denied. We will show that by the construction of FAM-programs, the possibility for conflict is elegantly handled so that exactly one of these two eventualities arises. Furthermore, we will show that the user's request is guaranteed to be processed in linear-time (data complexity). The same results apply to CAM-programs as well. The advantage with CAM-programs is that they are slightly easier to handle. Figure 2 shows the architecture of our system to facilitate this process.

The remainder of the paper is organized as follows. Section 2 discusses related work. The FAM architecture and the authorization specification language are described in section 3. Section 4 presents the details of the FAM- and CAM-programs. The contents of the FAM Policy Library are described in section 5. In section 6, we give an example to show how users can exploit the extensibility of FAM and specify their own protection policies for controlling accesses to the information owned by them. We conclude the paper in section 7.

2 Related Work

Access control in most commercial relational DBMSs is based on the System R authorization model [9, 11]. The System R model requires that all accesses that are to be allowed (we call these *positive* authorizations) must be specified, and enforces a *closed* policy: 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.

Recent authorization models also permit specification of *negative* authorizations stating accesses to be denied [5, 4, 7, 14]. Since these models allow both positive and negative authorizations to be specified, it is possible to have two authorizations, one stating that an access be allowed and the other stating that it be denied. In [5, 4], conflicts are resolved simply by adopting the *denials-take-precedence* policy, according to which the negative authorizations override the corresponding positive authorizations. Other models provide more sophisticated conflict resolution policies [7, 14, 16, 17].

Two models which permit multiple policies to be enforced are the Orion authorization model [16] and the model proposed in [3]. In addition to the positive and negative authorizations, these models permit authorizations that must be strongly obeyed and authorizations that allow for exceptions. Although these models are more general than the previous models, they suffer from the following drawbacks. First, the choices of policies that can be mapped on these models still remain limited. Second, these authorization models rely on the particular properties of the underlying data models (object-oriented data model and relational 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 with a wide variety of information sources (files, relations, objects, images, etc.).

Jonscher and Dittrich [12, 13] present an access control system for the protection of information in distributed federated database systems which allows the enforcement of different policies. Each policy is characterized by 19 attributes referring to different policy aspects (e.g., types of privileges allowed, signs of authorizations allowed, subjects' hierarchies to be considered). Different reference monitors are then defined, each enforcing a particular policy. The behavior of each access monitor is determined by the attributes specified for the policy implemented by the monitor. Each access request submitted to the system is forwarded to the responsible reference monitor for the object to be accessed and allowed or denied accordingly.

Finally, Woo and Lam [19] have used default logic to model authorization rules. Default logic is a very expressive framework. Unfortunately, default logic has several disadvantages over our framework. 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.

3 FAM Architecture and the Authorization Specification Language

The first component of the FAM architecture that we will discuss is the FAM authorization language. This language is a simple, logical language that uses a *fixed* set of predicates that we will define shortly to model a data system (that will also be defined below). The FAM authorization language allows the SSO to specify the "base" authorizations that s/he desires, as well as a set of authorization policies, by creating, explicitly or implicitly, a FAM-program (in the FAM authorization language).

1. The SSO may *explicitly* create one or more specifications in the FAM authorization language and include these specifications in a FAM-program.

2. The SSO may indicate that certain authorization policies in the authorization library should apply to an object (or a class of objects). In this case, s/he is *implicitly* specifying some rules in the FAM authorization language. The Authorization Library can *directly create* these rules and include them in the FAM-program the SSO is constructing.

The FAM-language contains three components: *first*, as this language must reason about data objects of different types (e.g. image files, text files, binary files, relations of different schemas, encapsulated objects, etc.) we need a formal definition of the data types involved, and this is contained in Section 3.1. Second, as these data objects will be manipulated by users who are usually grouped together into classes, we will define a *hierarchy* of users in Section 3.2. Finally, we will show how these two parameters (data objects and user hierarchies) lead to a simple language for expressing how users and groups are allowed to access these objects (Section 3.3).

3.1 Data Systems, Formalized

A *data system* **DS** consists of a 4-tuple $(\mathsf{Obj}, \mathsf{T}, \mathsf{S}, \mathsf{A})$ where

1. Obj is a set whose elements are called *objects*,

2. T is a set whose elements are called *types*,

3. $\mathsf{S} = \mathsf{U} \cup \mathsf{G}$ is a set whose elements are called *subjects* (U is the set of users and G is the set of groups), and

4. A is a set whose elements are called *authorization modes* or *actions*.

Here Obj is the set of objects containing information, access to which must be controlled. Types are named groups of objects; an object belongs to one and only one group.[1]

[1] The reason for this will become clear in Section 4.

Types could be defined in terms of the structures of a data model (e.g., tables in relational model), in terms of the kind of information or data contained in the objects (e.g., types `ps-files`, `tex-files`, `gif-files` could be defined as grouping all postscript, latex, and images files, respectively), or in terms of application requirements (e.g., in an organization types such as `Administrative-data`, `Budget-info`, `Letters` and `Invoices` could be defined).

Note that a data system makes no mention of which users are authorized to perform which accesses on which data objects.

Example 3.1 (File Systems) *A very simple example of a data system, which we call* $\mathbf{DS}_{file}$*, is a file system where* Obj *is the set of all files on the system,* T *is any user-specified classification of files (e.g. the type of a file might be the 3-letter extension of the file name, e.g. GIF, TIFF, TEX,etc.),* S *is the set of all user-ids of users allowed to log into that system and of groups into which they can be grouped, and* A *contains the actions* r, w, x *(read, write, execute).* □

Example 3.2 (Relational DBs) *A relational DB may be viewed as a data system* $\mathbf{DS}_{rel}$ *whose objects are the relations and tuples stored in the database, whose set of types is empty, and whose users are all the people allowed to access the database and groups possibly defined on them*[2] *and whose actions are composed of all operations executable on the relations (e.g., select,insert,update) plus the application programs users can invoke.* □

Example 3.3 (Object Oriented DBs) *An OO database system may be viewed as a data system* $\mathbf{DS}_{oo}$ *whose objects are all the object instances and their classes, whose subjects are users who can require execution of methods on objects and groups defined on them, and whose actions are all methods executable on the objects.*

□

In the sequel, we suppose that $\mathbf{DS} = (\mathsf{Obj}, \mathsf{T}, \mathsf{S}, \mathsf{A})$ is an arbitrary, but fixed data system. We will show how **DS** may be used to create an *authorization specification language* (**ASL**); authorizations specified in **ASL** may be used to determine whether or not a user's request to execute an action on an object may be allowed. The SSO may use a graphical user interface to specify authorizations in this language, as shown in Figure 2.

3.2 The User Hierarchy

Before specifying the details of **ASL**, we note that the subject of an authorization may be either an individual users or a *group* of users. A group may consist of individual users and other groups of users. These groups may well form a hierarchy. Figure 3 shows one such hierarchy specifying the groups to which different users belong.

Definition 3.1 (User Hierarchy) *Suppose* $\mathbf{DS} = (\mathsf{Obj}, \mathsf{T}, \mathsf{S}, \mathsf{A})$ *is a data system.* **DS** *is said to be* user-hierarchical *iff there exists a finite partially ordered set* $(\mathcal{G}, \leq)$ *such that:* x *is a* $\leq$*-minimal element of* $\mathcal{G}$ *iff* $x \in \mathsf{U}$.

Intuitively, the condition defining a user hierarchy says that the "bottom level" of the hierarchy consists of all individual users. For the example user hierarchy in Figure 3, the $\leq$-minimal elements are the users `Ann, Bob, Chris, Douglas, Ellen, Frank, Gary`, and `Henry`.

[2] Most database system supports at least one group, `public`, to which everybody belongs.

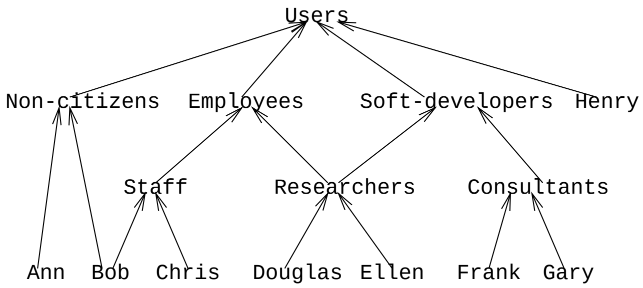

Figure 3: An Example User Hierarchy

Note that a subject can belong to several groups and, therefore, groups do not need to be disjoint. We say that the membership of a subject s in a group G is *direct* if the subject is defined as a member of the group. It is *indirect* if there exists a sequence of subjects $s_1, \ldots, s_n, n \geq 1$ such that either $n = 1$ or 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.

Throughout the rest of this paper, we will deal with user-hierarchical data systems. Note that a data system with no hierarchy is actually captured as a data system with a hierarchy $(G, \leq)$ where $G = \mathsf{U}$ and $x \leq y$ iff $x = y$. Thus, user-hierarchical data systems are more general than data systems that lack hierarchy. Thus, in the rest of this paper, $\mathbf{DS} = (\mathsf{Obj}, \mathsf{T}, \mathsf{S}, \mathsf{A})$ will denote an arbitrary, but fixed data system, which is user hierarchical via $(\mathcal{G}, \leq)$.

3.3 Authorization Specification Language (ASL)

In this section we introduce the basic constructs of the authorization language. As the main aim of **ASL** is to express authorization policies and as we have not defined this concept yet, we start by defining an authorization policy.

Definition 3.2 (Authorization Policy) *An* authorization policy *is a mapping that maps triples (o, u, a) consisting of an object, user, and action, respectively to the set* {*authorized, 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 $\mathsf{Obj} \cup \mathsf{T} \cup \mathsf{S} \cup \mathsf{A} \cup \mathsf{SA}$.

2. **Variable Symbols:** There are four sets V_o, V_s, V_a, V_{sa} of variable symbols ranging over the sets $\mathsf{Obj}, \mathsf{S}, \mathsf{A}, \mathsf{SA}$, respectively.

   In future, the following terminology will be used. Variables in V_o and members of Obj are object terms. Variables in V_s and members of S are subject terms. Variables in V_a and members of A are action terms. Variables in V_{sa} and members of SA are signed action 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. *Note that the mere presence of* $\mathtt{cando}(\mathtt{o}, \mathtt{s}, +\mathtt{a})$ *does not mean that subject* `s` *(members of* `s` *in case* `s` *is a group) will be allowed to perform action* `a` *on object* `o`. *It merely means that the SSO wants* `s` *to be able to perform action* `a` *on object* `t`. *Whether this will in fact be* <u>*allowed*</u> *depends upon whether there are any (explicit or derived) authorizations to the contrary.*

   (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 [1]). As in the case of `cando`, the fact that $\mathtt{dercando}(\mathtt{o}, \mathtt{u}, +\mathtt{a})$ holds does not mean that subject `s` (its members if `s` is a group) will be allowed to perform action a on object o.

   (c) A ternary predicate symbol, `do`, with the same arguments as `cando`. The predicate represents the accesses to be allowed or denied to each subject on each object. Intuitively, it enforces the access control policy. *If* $\mathtt{do}(\mathtt{o}, \mathtt{u}, +\mathtt{a})$ *is true, then user* `u` *will be allowed to perform action* `a` *on object* `o`. *Similarly, if* $\mathtt{do}(\mathtt{o}, \mathtt{u}, -\mathtt{a})$ *is true, then user* `u` *will not be allowed to perform action* `a` *on object* `o`. Note the distinction between `do`, `dercando` and `cando`.

   (d) A ternary predicate symbol, `done`. The first argument of `done` is an object term, the second is a subject term, and the third is an authorization term. *Intuitively, if* $\mathtt{done}(\mathtt{o}, \mathtt{u}, \mathtt{a})$ *is true, then user* `u` *had previously executed action* `a` *on object* `o`.

   (e) A ternary predicate symbol, `error`. The first argument of `error` is an object term, the second is a subject term, and the third is an authorization term. *Intuitively, if* $\mathtt{error}(\mathtt{o}, \mathtt{s}, \mathtt{a})$ *is true, then there is an error with respect to subject* `s`, *action* `a`, *and object* `o`.

   (f) A binary predicate `dirin` that takes as arguments, two subjects s_1, s_2. It captures the direct membership relationship between subjects (i.e., users and groups).

   (g) A binary predicate `in` that takes as arguments, two subjects s_1, s_2. It captures the indirect membership relationship.

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

   (i) A binary predicate `owner` whose first argument is an object term and second argument is a subject term. It associates a *unique* user (not group) with each object, called the *owner* of that object.

In the following we use the term *literal* to denote a predicate or its negation. For instance, if OT, ST, SAT are object terms, subject terms, and signed action terms respectively, then $\mathtt{cando}(OT, ST, SAT)$, $\neg\mathtt{cando}(OT, ST, SAT)$ are examples of literals.

Note that, in case s is a group, the fact that s has a positive (or negative) authorization for an action on an object does not necessarily imply that each member of s will have the positive (or negative) authorization for the access. Whether the authorizations specified for a group hold for a member m of the group depend on authorizations specified for m or for other groups to which m belongs.

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

Definition 3.3 (Done Rule) *A* done rule *is a rule of the form:*

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

where $o, s,$ and a are elements of $\mathsf{Obj}, \mathsf{S},$ *and* A *respectively. Note that "done" rules are facts, as their body is always empty.*

Done rules represent the accesses executed by users. They can be specified only by the system upon execution of accesses. These rules are useful for implementing those policies in which future accesses of the users are based on the accesses each user has exercised in the past (as in the case of the Chinese Wall policy [6]).

Definition 3.4 (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 $n \geq 0$, $< sign >$ is either $+$ or $-$ and, for each $0 < i \leq n$, L_i, either a `in` *a* `dirin`*,* `typeof` *or an* `owner` *literal is true.*

Authorization rules are specified by the SSO to allow or deny accesses to subjects. As stated before, authorization rules can be either *explicitly specified* by the SSO or *implicitly specified* through usage of the authorization library. The latter case will be discussed in Section 5 where we will discuss the construction of FAM-programs. 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 3.4 *Consider the following authorization rules.*

$$\begin{array}{rcl} \texttt{cando}(\texttt{file1}, \texttt{Employees}, +\texttt{write}) & \leftarrow & . \\ \texttt{cando}(o, \texttt{Ann}, -\texttt{write}) & \leftarrow & \texttt{typeof}(o, \texttt{Letters}). \\ \texttt{cando}(\texttt{file1}, s, +\texttt{read}) & \leftarrow & . \end{array}$$

The first rule states that the group `Employees` *is authorized to write file* `file1`*. The second rule states that* `Ann` *cannot write objects of type* `Letters`*. Finally, the third rule states that every subject can read* `file1`*.* ☐

The reader may wonder: if an authorization has already been specified for a group (e.g., `Employees`), surely this applies to all members of the group. If so, then why are statements involving `dirin` and `in` allowed in the bodies of rules? 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 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 subjects belonging to the group. To illustrate, consider the rule "$\texttt{cando}(\texttt{file1}, s, +\texttt{write}) \leftarrow \texttt{in}(s, \texttt{Employees})$".
This rule may be used to derive facts of the form
$\texttt{cando}(\texttt{file1}, \texttt{m}, +\texttt{write})$ for every member `m` of `Employees`. This rule may be used instead of specifying 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 by the SSO.* (See section 5).

Example 3.5 *Consider the following rules*

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

The first rule states that 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`*.* ☐

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 the authorizations explicitly stated by the administrator 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 3.5 (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 $< sign >$ is either $+$ or $-$ and $L_1, \ldots, L_n$ are either `cando`*,* `dercando`*,* `done`*,* `in`*,* `dirin`*,* `typeof` *or* `owner` *literals. All* `dercando`*-literals appearing in the body of a derivation rule must be positive.*

Example 3.6 *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 groups to their members. ☐

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 of 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 3.7 *Consider the following derivation rules:*

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

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 authorization. Likewise, `cando` and `dercando` rules may not imply either a positive or a negative authorization for a given object, subject, and authorization. The concept of a *decision rule*, given below, forces a decision to be made. Later, when we define a CAM-program, we will notice that they always guarantee *completeness* (for every object, subject, and authorization, either a positive authorization or a negative authorization is entailed by the CAM-program) and *consistency* (for every object, subject, and authorization, both positive and negative authorizations cannot be simultaneously entailed).

Definition 3.6 (Decision Rule) *A* decision rule *is a rule of the form*

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

where $L_1, \ldots, L_n$ are `cando`, `dercando`, `in`, `dirin`, `done`, `typeof`, `owner` *literals and every variable that appears in any of the L_i's also appears in the head of this rule.*

A decision rule states that a subject must be allowed, or forbidden, to exercise an authorization type on an object. Intuitively, `do` rules model access decisions of the system on the user requests to access objects.

The restriction on variables in the above definition is necessary to ensure that a set of decision rules will yield consistent access decision. This will be formally established in Theorem 4.2.

Note that there is a difference between a `do` rule and a `cando` rule. `cando` rules are authorizations, either positive or negative, specified by the SSO. These authorizations may conflict and, therefore, may not be obeyed by the system. *In contrast,* `do` *rules state what the system must do in response to an access request on the basis of the existing authorizations, specified or derived.*

Example 3.8 *Consider the following rules:*

$$\begin{aligned}
\texttt{do}(\texttt{file1}, s, +a) &\leftarrow \texttt{dercando}(\texttt{file1}, s, +a).\\
\texttt{do}(\texttt{file2}, s, +a) &\leftarrow \texttt{dercando}(\texttt{file2}, s, +a)\\
&\qquad \& \neg\texttt{dercando}(\texttt{file2}, s, -a).\\
\texttt{do}(o, s, +\texttt{read}) &\leftarrow \neg\texttt{dercando}(o, s, +\texttt{read})\\
&\qquad \& \neg\texttt{dercando}(o, s, -\texttt{read})\\
&\qquad \& \texttt{typeof}(o, \texttt{Pblc-docs}).
\end{aligned}$$

The first rule states that a subject can exercise an access on object `file1` *if s/he has a positive authorization for it. The second rule states that a subject can exercise an access on* `file2` *only if s/he both has a positive authorization for it and does not have a negative authorization for it. The last rule states that if no authorization has been specified for a subject for an object of type* `Pblc-docs`, *the subject can read the object.* □

Authorization, derivation, and decision rules defined above are all we need to specify authorizations and access control decisions. Our language supports an additional type of rule, called *integrity* rule, by which the SSO can define constraints that must hold on 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:*

$$\texttt{error}(o, s, a) \quad \leftarrow \quad L_1\& \ \ldots \& \ L_n.$$

where $L_1, L_2, \ldots, L_n$ are `cando`, `dercando`, `done`, `do`, `in`, `dirin`, `typeof`, `owner` *literals.*

The above rule derives an error every time the conditions in the right hand side of the rules are satisfied. 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 explicitly 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`".

Example 3.9 *Consider the following integrity rules*

$$\begin{aligned}
\texttt{error}(o, s, a) &\leftarrow \texttt{cando}(o, s, +a)\& \ \texttt{cando}(o, s, -a).\\
\texttt{error}(o, s, a) &\leftarrow \texttt{in}(s, \texttt{Employees})\& \ \texttt{in}(s, \texttt{Consultants}).\\
\texttt{error}(o, s, a) &\leftarrow \texttt{do}(o, s, \texttt{write})\& \ \texttt{do}(o, s, \texttt{evaluate})\&\\
&\qquad \texttt{typeof}(o, \texttt{Tech-reports}).\\
\texttt{error}(o, s, a) &\leftarrow \texttt{done}(o, s, \texttt{read})\& \ \texttt{done}(o', s, \texttt{read})\&\\
&\qquad \texttt{typeof}(o, \texttt{Budget-A})\& \ \texttt{typeof}(o', \texttt{Budget-B}).\\
\texttt{error}(o, s, a) &\leftarrow \texttt{done}(o, s, \texttt{read})\& \ \texttt{done}(o', s, \texttt{read})\&\\
&\qquad \texttt{typeof}(o, \texttt{Budget-B})\& \ \texttt{typeof}(o', \texttt{Budget-A}).
\end{aligned}$$

The first rule states that, given a user, an object, and an authorization triple (o, s, a), the SSO cannot simultaneously specify both a positive and a negative authorization. The second rule states that a subject cannot belong to both `Employees` *and* `Consultants`. *The third rule returns an error if a subject is authorized to both write and evaluate an object of type* `Tech-reports` *(e.g., an author of a paper should not referee that paper). Note that this enforces separation of duty [8]. The last two rules return an error if a user who has read an object of type* `Budget-A` *wishes to read an object of type* `Budget-B` *or vice-versa.*[3] *Note that this enforces a dynamic separation of duty constraint, as required in the Chinese Wall Policy [6]. The difference between static and dynamic separation of duty constraints is this. When considering separation of duties, if the user cannot be authorized for the 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; the user can decide which of the two accesses s/he wants to make. When the user exercises one of the accesses, the error rule will automatically rule out the possibility for the user to exercise the other.* □

The table below describes the different types of rules we have defined so far.

[3] We assume that the evaluation of whether the insertion of the done rule for an access would arise an error is performed before actually granting the access.

Rule Head	Rule Body
`done`	body is empty.
`cando`	body only contains `in`, `dirin`, `typeof`, `owner` literals.
`dercando`	body only contains `in`, `dirin`, `typeof`, `cando`, `dercando`, `owner` literals. Occurrences of `dercando` literals must be positive.
`do`	body only contains `in`, `dirin`, `typeof`, `cando`, `dercando`, `owner` literals. All variables occurring in the body also occur in the head.
`error`	body only contains `in`, `dirin`, `typeof`, `cando`, `dercando`, `owner` literals.

4 Authorization specifications, FAM-programs, and CAM-programs

Given the aforementioned definitions, we can now define an authorization specification as follows.

Definition 4.1 (Authorization Specification) *An* authorization specification **AS** *consists of a set of authorization (*`cando`*), derivation (*`dercando`*), decision (*`do`*), and integrity (*`error`*) rules.*

Proposition 4.1 *Any authorization specification* **AS** *is a* stratified *datalog program. Hence, the well-founded semantics [18] and the stable model semantics [10] of datalog programs both characterize the meaning of* **AS***, and both characterizations lead to the same meaning. We will use $SEM(P)$ to denote the meaning of an authorization specification* **AS** *w.r.t. the stable/well-founded semantics.* □

The above result tells us that there is no need to "re-invent" the wheel and create a new semantics for authorization specifications, because these specifications are really a small subclass of the family of stratified logic programs, whose semantics are well known and well understood. Though there are many different semantics for datalog with negation, all the major semantics coincide on the class of stratified logic programs. By the above theorem, all authorization specifications are stratified datalog programs, and hence their semantics is very well understood.

However, some work still needs to be done. The reason for this is that authorization specifications may express inconsistency. Notice that from a logical point of view, $+a$ and $-a$ are two distinct constants, and there is no axiom telling us that they lead to a logical inconsistency. Certainly, classical logic does not allow us to derive an inconsistency from statements of the form $\texttt{do}(o, u, +a), \texttt{do}(o, u, -a)$. Nevertheless, we would like to ensure that authorization specifications do not entail both $\texttt{do}(o, u, +a)$ and $\texttt{do}(o, u, -a)$ for any triple (o, u, a). FAM-programs, which we will define shortly below, ensure that this can never happen.

If R is a rule, we use $Head(R)$ and $Body(R)$ to denote the head and the body of the rule, respectively.

Definition 4.2 (Weakly Applicable Rule) *Suppose (o, u, a) consists of an object, a user (not a group !) and an authorization type (unsigned). A decision rule R of the form $\texttt{do}(OT, ST, < sign > AT)$ is said to be* weakly applicable *to (o, u, a) iff there exists a substitution θ such that $OT\theta = o$, $ST\theta = u$ and $AT\theta = a$.*

Note that to check if a rule R is weakly applicable to (o, u, a), we only check to see if the head of the rule has (o, u, a) as a ground instance, ignoring the sign in the authorization term.

Example 4.1 *Consider the three decision (*`do`*) rules of Example 3.8. Suppose we consider the triple* (`file1`,`john`,`read`). *There are two decision rules from Example 3.8 that are weakly applicable to this triple, viz. the first and third rules of Example 3.8. Note that weak applicability does not necessarily mean that the body of the rule is true (which is why we use the expression "weak applicable" rather than "applicable").*

Note, in particular, that if a decision rule R is *not* weakly applicable to (o, u, a), then there is no chance that R can be used to derive either a positive authorization or a negative authorization of a for user u on object a.

Definition 4.3 (Complementary Literals) *Two literals L_1, L_2 are said to be* complementary *iff either of the conditions listed below holds:*

1. *L_1 is the atom A and $L_2 = \neg A$, or vice-versa;*
2. *$L_1 = \texttt{typeof}(o, t_1)$ and $L_2 = \texttt{typeof}(o, t_2)$ and $t_1 \neq t_2$.*
3. *$L_1 = \texttt{owner}(o, u_1)$ and $L_2 = \texttt{owner}(o, u_2)$ and $u_1 \neq u_2$.*

The second and third rules state, respectively, that each object has exactly one type and one owner.

Definition 4.4 (Clash-Free Decision Rules) *A set DR of decision rules is said to be* clash-free *iff for every triple (o, u, a) consisting of an object, a user (not a group !) and an authorization type (unsigned), it is the case that: if $R_1, R_2 \in DR$ are weakly applicable to (o, u, a) and if $Head(R_1) = \texttt{do}(OT_1, ST_1, +AT_1)$ and $Head(R_2) = \texttt{do}(OT_2, ST_2, -AT_2)$ and (OT_1, ST_1, AT_1) and (OT_2, ST_2, AT_2) are unifiable via most general unifier θ, then $(Body(R_1)\theta \;\&\; Body(R_2)\theta)$ contains a pair of complementary literals.*

For example, consider the three rules given below:

$$
\begin{array}{lll}
1: \texttt{do(file1, X, +read)} & \leftarrow & \texttt{dirin(X, Employees)}. \\
2: \texttt{do(file1, X, -read)} & \leftarrow & \neg\texttt{dirin(X, Employees)}\ \& \\
 & & \texttt{dirin(X, Consultants)}. \\
3: \texttt{do(file1, X, -read)} & \leftarrow & \texttt{dirin(X, Policeman)}.
\end{array}
$$

The sets of rules $\{1, 2\}$ and $\{2, 3\}$ are clash-free. However, the set $\{1, 3\}$ is not-clash free. The reason is that rules (1) and (3) may jointly imply that a person who is both an employee and a policeman has both positive and negative read access to file1. This is a potential clash.

NOTE: The reader should note that if a set of rules is clash free according to definition 4.4 above, then we are guaranteed by Theorem 4.2 that no inconsistencies will occur. The converse is not necessarily true. In particular, there may exist inconsistency-free authorization specifications that do not satisfy the clash free condition. However, as Theorem 4.3 below shows, (a subclass of) clash-free authorization specifications are enough to represent all authorization policies. *For purposes of efficient implementation, the condition in definition 4.4 is very easy to check, and all authorization policies can be expressed in this framework. Thus, clash-free authorization specifications are adequate for all purposes pertaining to articulation and implementation of authorization policies.*

4.1 FAM-Programs

We now come to the important definition of FAM-programs. Note that an SSO creates FAM-programs either directly, or using a GUI front-end as shown in Figure 2. Furthermore, all components of the Authorization Library (to be discussed in Section 5) are also FAM-programs.

Definition 4.5 (FAM-program) *An authorization specification* **AS** *is said to be a* FAM-program *iff the set of decision (*`do`*) rules in* **AS** *is:*

1. *clash-free and*
2. *for each triple (o, u, a) of object, user, and authorization type, there exists at least one weakly applicable* `do`*-rule in* **AS**.

Thus, FAM-programs are authorization specifications whose `do`-rules are clash free and that satisfy one extra condition listed above. When the SSO specifies authorizations, the system needs to check whether the resulting authorization specification is indeed a FAM-program (if not, a syntax error needs to be reported). The second condition in the above definition guarantees that at least one rule in the FAM-program will apply to any user access request. Other systems, such as Orion, also have a similar requirement. We will show later that, using a more tightly specified class of authorization specifications, we can eliminate this requirement. The following result shows that checking if an authorization specification is a FAM-program can be computed in polynomial time.

Theorem 4.1 *The problem of determining whether an authorization specification,* **AS***, is a FAM-program can be solved in time $O(n \times k^2)$ where n is the number rules in* **AS** *and k is the* maximum *number of literals in the body of a rule. (Typically, k is small).* □

The following result tells us that FAM-programs give us *at most one* authorization (positive or negative) for each object/user/authorization type triple. No inconsistencies arise.

Theorem 4.2 *Suppose P is a FAM-program, and (o, u, a) is a triple consisting of an object, user, and authorization type, respectively. Then: $SEM(P)$ entails at most of the two literals $\mathtt{do}(o, u + a)$, $\mathtt{do}(o, u, -a)$.* □

Note that the above theorem would not hold if the syntax of the decision rules did not contain the restriction that variables appearing in the body of the decision rule must also appear in the head.

The following result tells us that FAM-programs are expressive enough to express all authorization policies.

Theorem 4.3 *Suppose γ is any authorization policy. Then there exists a FAM-program P_γ such that:*

1. *$\gamma(o, u, a) =$ authorized iff $\mathtt{do}(o, u, +a)$ is true according to the semantics of F_γ.*
2. *$\gamma(o, u, a) =$ denied iff $\mathtt{do}(o, u, -a)$ is true according to the semantics of F_γ.* □

According to the above result, if P is a FAM-program, then the authorization policy expressed by P coincides with the *materialization* of the query `do(Obj, User, Auth)`. *The other predicates need not be explicitly materialized.* The following result follows immediately from well known results on the data complexity of stratified datalog programs.

Proposition 4.2 *Suppose P is a FAM-program, expressing an authorization policy γ. The problem "given as input a triple (o, u, a), determine whether this access is authorized or denied" can be solved in linear time w.r.t. the number of rules in P.* □

4.2 CAM-programs

FAM-programs have a drawback: the SSO has to explicitly specify both positive accesses and negative accesses through the `do`-predicate. However, this may, on occasion, be rather cumbersome and inconvenient. In order to remedy this problem, we introduce the notion of a CAM-program below.

Definition 4.6 (CAM-program) *An authorization specification* **AS** *is said to be a* semi-CAM program *iff each* `do` *rule in* **AS** *has a head of the form $\mathtt{do}(OT, ST, +AT)$, i.e. the* sign *attached to the head of each* `do` *rule in* **AS** *is positive, negative signs are forbidden.*

A CAM-program *is a semi-CAM program together with the single extra rule:*

$$\mathtt{do}(O, U, -A) \quad \leftarrow \quad \neg\mathtt{do}(O, U, +A).$$

The special rule contained above in a CAM-program states that if a positive authorization is not implied, then the authorization may be assumed to be negative. *It is easy to see that CAM-programs are not stratified programs as in the case of FAM-programs, because of the insertion of this extra rule.* However, the following theorem tells us that CAM-programs still have all the nice, desirable properties of FAM-programs.

Proposition 4.3 *Every CAM-program is a* locally stratified *(cf. [15]) datalog program, and hence, it is guaranteed to possess a unique stable model, which coincides with the well-founded semantics of P.*

Thus, even though CAM-programs are not stratified, they are "locally stratified" (cf. [15]) and are guaranteed to possess the nice properties that FAM-programs have. Furthermore, the following result tells us that CAM programs and FAM-programs are equally expressive, and hence, as a corollary, every authorization program can be captured as a CAM-program.

Theorem 4.4 *Let us suppose that (o, u, a) is an object-user-authorization triple.*

1. *If P is a FAM program, then there exists a CAM program Q such that for each triple (o, u, a):*
   - *(a) $(o, u, +a)$ is true according to the semantics of P iff $(o, u, +a)$ is true according to the semantics of Q.*
   - *(b) $(o, u, -a)$ is true according to the semantics of P iff $(o, u, -a)$ is true according to the semantics of Q.*
2. *Conversely, if Q is a CAM program, then there exists a FAM program P satisfying the two conditions in the preceding item.*
3. *If Q is a CAM program, then either $(o, u, +a)$ is true according to the semantics of P or $(o, u, -a)$ is true according to the semantics of P, but not both.*

Finally CAM-programs are complete in the sense that given a CAM program P, and any triple (o, u, a), then the semantics of P makes exactly one of $(o, u, +a)$, $(o, u, -a)$ true. CAM-programs can also express *all* authorization policies. However, CAM-programs are complete and also often more convenient from the user's point of view – in particular, the SSO only needs to specify positive `do`-rules, and does not, therefore, have to check the clash-free condition, and the completeness condition inherent in the definition of FAM-programs. *All other results in Section 4.1 apply to CAM-programs as well. For reasons of space, we are not explicitly re-stating them here.*

5 The FAM Policy Library

As the reader may have noticed from Figure 2, the SSO may set up authorizations by interacting with a GUI front end that will create FAM-programs for him. To do this, the user may pick a subject, an object, and an authorization type, and request that a particular policy from the Authorization Library be "applied" to that subject, object, and authorization type. The purpose of this section is two-fold: first, we show that the language of FAMs is rich enough to express a wide variety of authorization policies in the literature, and second, we show the content of the Authorization Library. In particular we are interested in policies for dealing with positive and negative authorizations with respect to subject hierarchies.

From the point of view of authorization specification, we identify three main categories of policies:

Closed Only positive authorizations can be specified. A user is permitted an access only if s/he or any of the groups to which s/he belongs have been granted an authorization for it.

Open Only negative authorizations can be specified. A user is permitted an access only if neither s/he nor any of the groups to which s/he belongs have been granted a negative authorization for it.

Hybrid Both positive and negative authorizations can be specified. How authorizations propagate from the groups to their members and how conflicts are possibly resolved depend on policy decisions. This category therefore covers a variety of different policies.

5.1 Hybrid policies

In the case of hybrid policies different approaches can be taken to regulate the coexistence of negative and positive authorizations. We can identify two basic components in hybrid policies: the *derivation* component that determines how authorizations of groups propagate to their members and the *conflict resolution* component that determines how possible conflicts between authorizations are solved. We discuss possible approaches that can be taken for each of these components.

Derivation When authorizations specified for groups propagate to their members and when a subject belongs, directly or indirectly, to groups with conflicting (i.e., positive and negative) authorizations for the same access, then an inconsistency may arise. Derivation rules determine which authorization, if any, should override the other. The following choices are possible:

No overriding (NoOver) : Authorizations of groups always propagate to their members regardless of the possibility of conflicts among authorizations.

Subgroups overriding (SubOver) : If conflicting authorizations propagate to a member, authorizations given to a subgroup override the authorizations given to a supergroup.

Path overriding (PathOver) : If conflicting authorizations propagate to a member because of his/her membership in two groups G and G', authorizations of a subgroup G override the authorizations of a supergroup G' if there is paths from the member to G' that passes from G.

Conflict resolution When both a positive and a negative authorization are present (explicit or derived) for a subject, object, and access mode, then conflict resolution rules must be devised to determine which of the two authorizations should take precedence over the other. In other words, the conflict resolution rules define whether the access must be denied or authorized. The following solutions are possible:

No conflicts allowed (NoCon) : The presence of both a positive and a negative authorization for a subject is considered an inconsistency and it is therefore not accepted.

Permissions take precedence(Perm) : The positive authorization takes precedence over the negative one. In this case the access is authorized.

Denials take precedence (Denials) : The negative authorization takes precedence over the positive one. In this case the access is denied.

The combination of the different approaches allows us to identify eight different hybrid policies, as illustrated in Table 1.

5.2 Expressing policies in ASL

We now show how each of the above policies can be represented by the use of a few rules in our language (these rules may easily be stored in the Authorization Library). In the case of hybrid policies, we give the rules corresponding to each possible approach that can be taken for derivation and conflict resolution. Rules enforcing a specific policy are given by the union of the rules implementing the specific choice.

Closed policy In the implementation of the closed policy, the SSO can specify only positive `cando` facts. A user is authorized for an access only if a positive cando fact has been specified for the user or for any of the members to which he belongs. Specification of negative `cando` facts is considered an inconsistency and is therefore rejected. This is expressed by the following rules.

$$
\begin{aligned}
\texttt{dercando}(o,u,+a) &\leftarrow \texttt{cando}(o,s,+a)\&\ \texttt{in}(u,s).\\
\texttt{do}(o,u,+a) &\leftarrow \texttt{dercando}(o,u,+a).\\
\texttt{error}(o,s,a) &\leftarrow \texttt{cando}(o,s,-a).\\
\texttt{do}(o,u,-a) &\leftarrow \neg\texttt{do}(o,u,+a).
\end{aligned}
$$

Conflict resolution	Derivation: No overriding	Subgroup overriding	Path overriding
No conflicts	not applicable	SubOver-NoCon	PathOver-NoCon
Permissions take prec.	NoOver-Perm	SubOver-Perm	PathOver-Perm
Denials take prec.	NoOver-Denials	SubOver-Denials	PathOver-Denials

Table 1: Hybrid policies

The `error` rule is used to signal inconsistency upon insertion of a negative authorization and therefore reject it. Note that is it is not strictly necessary to have it. Indeed negative authorizations, even if inserted, are ignored by the `do` rules implementing the policy.

Open policy This policy is dual to the previous one. The SSO can specify only negative `cando` facts. A user is authorized for an access only if no negative cando fact has been specified for the user or for any of the members to which s/he belongs. Specification of positive `cando` facts is considered an inconsistency and it is therefore rejected. This is expressed by the following rules.

$$\begin{aligned}
\texttt{dercando}(o,u,-a) &\leftarrow \texttt{cando}(o,s,-a)\&\ \texttt{in}(u,s).\\
\texttt{do}(o,u,+a) &\leftarrow \neg\texttt{dercando}(o,u,-a).\\
\texttt{error}(o,s,a) &\leftarrow \texttt{cando}(o,s,+a).\\
\texttt{do}(o,u,-a) &\leftarrow \neg\texttt{do}(o,u,+a).
\end{aligned}$$

As in the previous case, the error rule is not strictly necessary since possible positive authorizations are ignored by the decision rules.

As for the hybrid policies, modular rules provide the behavior of each specific policy decision for the derivation of authorizations and for conflict resolution. Each specific policy is then obtained by the union of the rules enforcing the corresponding policy components.

Rules for enforcing derivation are as follows:

No overriding

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

The rules derive a negative/positive `dercando` fact for an access for a subject whenever a negative/positive `cando` fact exists for a group to which the subject belongs.

Subgroup overrides

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

The rules derive a negative/positive `dercando` fact for an access for a subject if the subject: (*i*) belongs to a group s' for which a negative/positive `cando` fact for the access has been specified and, (*ii*) does not belong to any group s'', subgroup of s', for which a positive/negative `cando` fact for the access has been specified.

Subgroup overrides along a path

$$\begin{aligned}
\texttt{dercando}(o,s,+a) &\leftarrow \texttt{cando}(o,s,+a).\\
\texttt{dercando}(o,s,-a) &\leftarrow \texttt{cando}(o,s,-a).\\
\texttt{dercando}(o,s,+a) &\leftarrow \texttt{dercando}(o,s',+a)\ \&\\
&\quad\ \neg\texttt{cando}(o,s,-a)\ \&\ \texttt{dirin}(s,s').\\
\texttt{dercando}(o,s,-a) &\leftarrow \texttt{dercando}(o,s',-a)\ \&\\
&\quad\ \neg\texttt{cando}(o,s,+a)\&\ \texttt{dirin}(s,s').
\end{aligned}$$

The rules work by recursively propagating authorizations from a group to its direct subgroups. The first two rules derive a `dercando` fact for any `cando` fact specified. The last two rules enforce propagation of authorizations to users who may belong to multiple groups. According to these rules, an authorization specified for a group propagates to the members of the direct subgroup if it there is no explicit conflicting authorizations for the subgroup.

The rules for enforcing conflict resolution are as follows:

No conflicts

$$\begin{aligned}
\texttt{do}(o,u,+a) &\leftarrow \texttt{dercando}(o,u,+a).\\
\texttt{error}(o,u,a) &\leftarrow \texttt{dercando}(o,u,+a)\ \&\\
&\quad\ \texttt{dercando}(o,u,-a).\\
\texttt{do}(o,u,-a) &\leftarrow \neg\texttt{do}(o,u,+a).
\end{aligned}$$

The first rule states that an access is allowed only if there is a positive `dercando` fact for it. The `error` rule signals inconsistency if both a positive and a negative `dercando` fact exist for an access.

Denials take precedence

$$\begin{aligned}
\texttt{do}(o,u,+a) &\leftarrow \texttt{dercando}(o,u,+a)\ \&\\
&\quad\ \neg\texttt{dercando}(o,u,-a).\\
\texttt{do}(o,u,-a) &\leftarrow \neg\texttt{do}(o,u,+a).
\end{aligned}$$

The rule states that an access is authorized only if there exists a positive `dercando` fact and there is no negative `dercando` fact for it.

Permissions take precedence

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

An access is allowed only if there is a positive `dercando` fact for it.

1	$\texttt{do}(o, u, \texttt{+read}) \leftarrow \neg\texttt{cando}(o, s, \texttt{-read})\ \&\ \texttt{in}(u, s)\ \&\ \neg\texttt{in}(u, \texttt{Non-citizens})\ \&\ \texttt{typeof}(o, \texttt{Nat-docs}).$ $\texttt{do}(o, u, \texttt{+read}) \leftarrow \texttt{cando}(o, s, \texttt{+read})\ \&\ \texttt{in}(u, s)\ \&\ \texttt{in}(u, \texttt{Non-citizens})\ \&\ \texttt{typeof}(o, \texttt{Nat-docs})$ $\texttt{do}(o, u, \texttt{+write}) \leftarrow \texttt{cando}(o, u, \texttt{+write})\ \&\ \texttt{typeof}(o, \texttt{Nat-docs}).$
2	$\texttt{do}(o, u, \texttt{+}a) \leftarrow \neg\texttt{cando}(o, s, \texttt{-}a)\ \&\ \texttt{in}(u, s)\ \&\ \texttt{typeof}(o, \texttt{Pbl-info}).$
3	$\texttt{dercando}(o, s, \texttt{+}a) \leftarrow \texttt{cando}(o, s', \texttt{+}a)\&\ \neg\texttt{cando}(o, s'', \texttt{-}a)\&\ \texttt{in}(s, s')\&\ \texttt{in}(s, s'')\&\ \texttt{in}(s'', s')\&\ s'' \neq s'\ \&\ \texttt{typeof}(o, \texttt{Projects-info}).$ $\texttt{dercando}(o, s, \texttt{-}a) \leftarrow \texttt{cando}(o, s', \texttt{-}a)\&\ \neg\texttt{cando}(o, s'', \texttt{+}a)\&\ \texttt{in}(s, s')\&\ \texttt{in}(s, s'')\&\ \texttt{in}(s'', s')\&\ s'' \neq s'\ \&\ \texttt{typeof}(o, \texttt{Projects-info}).$ $\texttt{do}(o, u, \texttt{+}a) \leftarrow \texttt{dercando}(o, u, \texttt{+}a)\ \&\ \texttt{typeof}(o, \texttt{Projects-info}).$
4	$\texttt{dercando}(o, s, \texttt{+}a) \leftarrow \texttt{cando}(o, s', \texttt{+}a)\&\ \neg\texttt{cando}(o, s'', \texttt{-}a)\&\ \texttt{in}(s, s')\&\ \texttt{in}(s, s'')\&\ \texttt{in}(s'', s')\&\ s'' \neq s'\ \&\ \texttt{typeof}(o, \texttt{Budget-info}).$ $\texttt{dercando}(o, s, \texttt{-}a) \leftarrow \texttt{cando}(o, s', \texttt{-}a)\&\ \neg\texttt{cando}(o, s'', \texttt{+}a)\&\ \texttt{in}(s, s')\&\ \texttt{in}(s, s'')\&\ \texttt{in}(s'', s')\&\ s'' \neq s'\ \&\ \texttt{typeof}(o, \texttt{Budget-info}).$ $\texttt{do}(o, u, \texttt{+}a) \leftarrow \texttt{dercando}(o, u, \texttt{+}a)\&\ \neg\texttt{dercando}(o, u, \texttt{-}a)\ \&\ \texttt{typeof}(o, \texttt{Budget-info}).$
5	$\texttt{do}(o, u, \texttt{+read}) \leftarrow \texttt{cando}(o, s, \texttt{+read})\ \&\ \texttt{in}(u, s)\ \&\ \texttt{typeof}(o, \texttt{Tech-reports}).$ $\texttt{do}(o, u, \texttt{+write}) \leftarrow \texttt{owner}(o, u)\ \&\ \texttt{typeof}(o, \texttt{Tech-reports}).$
6	$\texttt{do}(o, u, \texttt{+}a) \leftarrow \texttt{owner}(o, u)\ \&\ \texttt{typeof}(o, \texttt{Prvt-docs}).$
CAM-completeness	$\texttt{do}(o, u, \texttt{-}a) \leftarrow \neg\texttt{do}(o, u, \texttt{+}a).$

Figure 4: CAM rules for the example

6 An Example

As stated earlier, a major advantage of using our framework is that users are not constrained to the use of a single policy. They can exploit the extensibility of FAM and specify their own protection policies for controlling accesses to the information owned by them.

To illustrate this, suppose the SSO has to specify authorizations on a collection of objects of different types and has different access control requirements, as described below. For the sake of simplicity we consider only two authorization types: `read` and `write`. Moreover, we omit the specification of integrity rules whenever they are not strictly necessary.

1. Type `Nat-docs`: All documents having this type can be freely read by those users that are citizens unless an exception is stated (*open policy*). By default foreigners cannot read a document. A foreigner will be allowed to read a document only if there is a positive authorization for the access (*closed policy*). Write operations are allowed only to those users explicitly authorized as individuals, i.e., authorizations given to groups do not apply (*a restricted form of closed policy*).

2. Type `Pbl-info`: It includes all documents freely available to everybody for reading, and also for writing unless explicitly denied (*open policy*).

3. Type `Projects-info`. Both positive and negative authorizations can be specified, thus allowing for exceptions. The authorizations specified for a subgroup take precedence over the authorizations specified for groups. A user can read a document if s/he owns a positive authorizations for it either personally or as member of some groups (*hybrid policy: Subgroup overrides - Permissions take precedence*).

4. Type `Budget-info`. Authorizations are specified and propagated to members of groups as in the previous case. A user can exercise an access only if s/he owns a positive authorization and does not own any negative authorization for it, i.e., in case of contrasting authorizations the negative authorization overrides the positive one (*hybrid policy: Subgroup overrides - Denials take precedence*).

5. Type `Tech-reports`. These documents are available for reading to whomever authorized (*open policy*). Write operations can be executed only by the owner.

6. Type `Prvt-docs`. It contains all documents that can be accessed only by the owner.

To enforce the above policies on the different types of objects, the SSO will need to specify the rules reported in Figure 4. Note that the rules in Figure 4 are only the derivation and decision rules of the policy. No authorization (`cando` rule) has been explicitly specified yet. If no explicit authorization is specified, each access decision will be based on the default decision stated by the `do` rules. For instance, a citizen will be always granted read access to a document of type `Nat-docs` and a foreigner will be always denied for it. The SSO can also specify cando rules stating which accesses s/he authorizes and which s/he does not. The access decisions will be based in this case on the application of the policy decision rules to the authorizations specified by the SSO. For instance, suppose a negative authorization is specified for `Gary` to read document `tax-report`. The system will behave as before for all the other users. `Gary`, as a citizen, will be allowed to read all documents of type `Nat-docs` but he will be denied for the read access to document `tax-report`.

We close this section with a final remark concerning authorization specification. In the example, we have assumed the SSO specifies all the rules. Note that in case a decentralized administration is preferred, single users will be allowed to specify rules on the objects they own. The only difference with respect to the centralized administration by the SSO is that in the case of decentralized administration, a literal $\texttt{owner}(o, \texttt{u})$ must appear in the body of each rule specified by user `u`, where o is the object term appearing in the head of the rule. In this way, rules specified by a user will be applicable only to the objects owned by this user.

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 introduced the important concept of FAM and CAM programs. A system security officer, using our FAM architecture, may specify an authorization for any object (or class of objects) and any user (or class of users), and then specify an access control policy, using our `do` predicate, specifying how authorizations will propagate from user groups to individual users. The user may select such policies from an authorization library, thus avoiding dealing with the syntax of FAM/CAM programs, or the user may choose to invent his/her own application specific policies, thus extending the library to suit his/her needs. We have shown that FAM and CAM programs have a well understood syntax and semantics (through the stratified, and locally stratified semantics, respectively), and that they are guaranteed to lead to one and only one authorization per user request.

This work is only a first step towards the definition of a unified framework for enforcing multiple access control policies and leaves space for further work. A first issue to be investigated concerns the enrichment of the model. In the paper we have assumed that an object can belong to at most one type. The reason for this was to consider two `typeof` predicates referring to different types as complementary; thus avoiding rules referred to objects of different types to clash. We note that the constraint on type uniqueness can be removed from the model. In this case two `typeof` literals referred to different types should not be considered complementary anymore. Further extensions concern the inclusion of authorizations for roles and role-related constraints and the investigation of administrative policies regulating insertion and deletion of rules from the FAM library.

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] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. A temporal access control mechanism for database systems. *IEEE Trans. on Knowledge and Data Engineering*, 8(1):67–80, February 1996.

[3] Elisa Bertino, Sushil Jajodia, and Pierangela 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] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. Authorizations in relational database management systems. In *Proc. ACM Conf. on Computer and Communications Security*, pages 140–150, Fairfax, VA, November 1993.

[5] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. An extended authorization model for relational databases. *IEEE Trans. on Knowledge and Data Engineering*, 9(1), 1997.

[6] 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.

[7] Hans H. Brüggemann. Rights in an object-oriented environment. In Carl E. Landwehr and Sushil Jajodia, editors, *Database Security, V: Status and Prospects*, pages 99–115. North-Holland, Amsterdam, 1992.

[8] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proc. Symp. on Security and Privacy*, pages 184–194, Oakland, CA, 1987.

[9] R. Fagin. On an authorization mechanism. *ACM Trans. on Database Systems*, 3(3):310–319, September 1978.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th Int'l. Conf. and Symp. on Logic Programming*, pages 1070–1080, 1988.

[11] P.G. Griffiths and B. Wade. An authorization mechanism for a relational database system. *ACM Trans. on Database Systems*, 1(3):243–255, September 1976.

[12] D. Jonscher and K. R. Dittrich. Argos - A configurable access control system for interoperable environments. In David L. Spooner, Steven A. Demurjian, and John E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 43–60. Chapman & Hall, London, 1996.

[13] Dirk Jonscher and Klaus R. Dittrich. An approach for building secure database federations. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 24–35, Santiego, Chile, 1994.

[14] 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.

[15] T. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of deductive databases*, pages 193–216. Morgan Kaufmann, San Mateo, 1988.

[16] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Trans. on Database Systems*, 16(1):89–131, March 1991.

[17] H. Shen and P. Dewan. Access control for collaborative environments. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, pages 51–58, November 1992.

[18] A. van Gelder. The alternating fixpoint of logic programs with negation. In *ACM Symp. on Principles of Database Systems*, pages 1–10, 1989.

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