Access Control Policies and Languages in Open Environments

S. De Capitani di Vimercati[1], S. Foresti[1], S. Jajodia[2], and P. Samarati[1]

[1] Università degli Studi di Milano
`{decapita, foresti, samarati}@dti.unimi.it`
[2] Center of Secure Information Systems
George Mason University
`jajodia@gmu.edu`

1 Introduction

Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied. Access control plays an important role in overall system security. The development of an access control system requires the definition of the regulations (*policies*) according to which access is to be controlled and their implementation as functions executable by a computer system. The access control policies are usually formalized through a security *model*, stated through an appropriate specification *language*, and then enforced by the access control *mechanism* enforcing the access control service. The separation between policies and mechanisms introduces an independence between protection requirements to be enforced on the one side, and mechanisms enforcing them on the other. It is then possible to: *i)* discuss protection requirements independently of their implementation, *ii)* compare different access control policies as well as different mechanisms that enforce the same policy, and *iii)* design mechanisms able to enforce multiple policies. This latter aspect is particularly important: if a mechanism is tied to a specific policy, a change in the policy would require changing the whole access control system; mechanisms able to enforce multiple policies avoid this drawback. The formalization phase between the policy definition and its implementation as a mechanism allows the definition of a formal model representing the policy and its working, making it possible to define and prove security properties that systems enforcing the model will enjoy [30]. Therefore, by proving that the model is "secure" and that the mechanism *correctly implements* the model, we can argue that the system is "secure" (*with respect to the definition of security considered* [37]).

The definition of access control policies (and their corresponding models) is far from being a trivial process. One of the major difficulty lies in the interpretation of, often complex and sometimes ambiguous, real world security policies and in their translation in well defined and unambiguous rules enforceable by a computer system. Many real world situations have complex policies, where access decisions depend on the application of different rules coming, for example, from laws, practices, and organizational regulations. A security policy must capture all the different regulations to be enforced and, in addition, must also consider possible additional threats due to the use of a computer system. Given the complexity of the scenario, there is a need for flexible, powerful, and expressive access control services to accommodate all the different requirements that may need to be expressed, while at the same time be simple both in terms of use (so that specifications can be kept under control) and implementation (so to allow for its verification).

An access control system should include support for the following concepts/features.

- *Expressibility.* An access control service should be expressive enough so that the policy can suit all the data owner's needs. To this purpose, several of the most recent language designs rely on concepts and techniques from logic, specifically from logic programming [16,28,32–34,48]. Logic languages are particularly attractive as policy specification languages (see Sect. 3). One obvious advantage lies in their clean and unambiguous semantics, suitable for implementation validation, as well as formal policy verification. Second, logic languages can be expressive enough to formulate all the policies introduced in the literature. The declarative nature of logic languages yields a good compromise between expressiveness and simplicity. Their high level of abstraction, very close to the natural language formulation of the policies, makes them simpler to use than imperative programming languages. However, security managers are not experts in formal logics, either, so generality is sometimes traded for simplicity.
- *Efficiency.* Access control efficiency is always a critical issue. Therefore, simple and efficient mechanisms to allow or deny an access are key aspects (see Sect. 3).
- *Simplicity.* One of the major challenges in the definition of a policy language is to provide expressiveness and flexibility while at the same time ensuring easiness of use and therefore applicability. An access control language should therefore be based on a high level formulation of the access control rules, possibly close to natural language formulation (see Sect. 4).
- *Anonymity support.* In open environments, not all access control decisions are identity-based. Resource/service requesters depend upon their *attributes* (usually substantiated by certificates) to gain accesses to resources (see Sect. 5).
- *Policy combination and conflict-resolution.* If multiple modules (e.g., for different authorities or different domains) exist for the specification of access control rules, the access control system should provide a means for users to specify how the different modules should interact, for example, if their union (maximum privilege) or their intersection (minimum privilege) should be considered (see Sect. 6). Also, when both permissions and denials can be specified, the problem naturally arises of how to deal with *incompleteness* (accesses for which no rule is specified) and *inconsistency* (accesses for which both a denial and a permission are specified). Dealing with incompleteness (requiring the authorizations to be complete would be very impractical) requires support of a default policy either

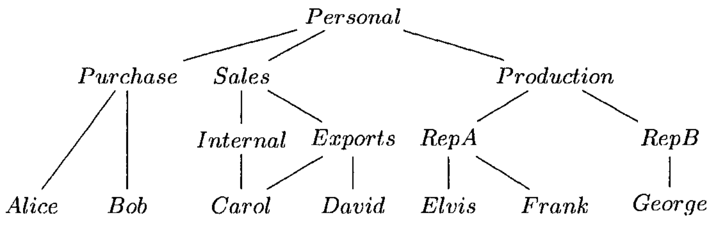


Fig. 1. An example of user-group hierarchy

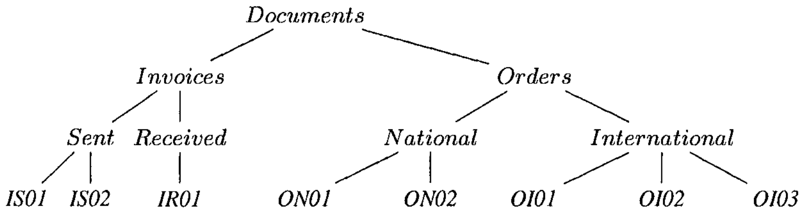


Fig. 2. An example of object hierarchy

supported by the system or specified by the users. Dealing with inconsistencies require support for conflict resolution policies.

In this chapter, after a brief overview of the basic concepts on which access control systems are based, we illustrate recent proposals and ongoing work addressing access control in emerging applications and new scenarios. The remainder of this chapter is structured as follows. Section 2 introduces the basic concepts of access control. Section 3 presents a logic-based framework for representing access control policies. Section 4 briefly describes some XML-based access control languages and illustrates the XACML policy model and language. XACML is a OASIS standard that provides a means for expressing and interchanging access control policies in XML. Section 5 introduces recent solutions basing the access control decisions on the evaluation of users' attributes rather than on their explicit identity. Section 6 addresses the problem of combining authorization specifications that may be independently stated. We describe the characteristics that a policy composition framework should have and illustrate some current approaches. Finally, Sect. 7 concludes the chapter.

2 Basic Concepts

A first step in the development of an access control system is the identification of the *objects* to be protected, the *subjects* that execute activities and request access to objects, and the *actions* that can be executed on the objects, and that must be controlled. More precisely, an access control system should support the following concepts.

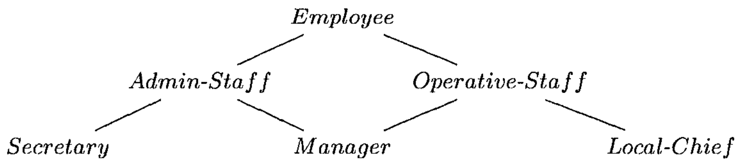


Fig. 3. An example of role hierarchy

- *Users* (U) are entities requesting access to objects. Abstractions can be defined within the domain of users. Intuitively, abstractions allow to define group of users. Users together with their groups, denoted G, define a partial order that introduces a hierarchy on the user domain. Figure 1 illustrates an example of user-group hierarchy.
- *Data Items* (Obj) are the objects of the system that have to be protected and on which access rights can be specified. Objects can be organized in a hierarchical structure, defining sets of objects that can be referred together with a given name. The definition of groups of objects (*object types*), denoted T, introduces a *hierarchy* of objects and groups thereof. For instance, a file system can be seen as an object hierarchy, where files are single objects and directories are groups thereof. Figure 2 illustrates an example of object hierarchy.
- *Access Types* (A) are the actions that can be executed on an object. The actions may vary depending on the kind of objects considered.
- *Roles* (R) are sets of privileges. A user playing a role has the ability to execute the privileges associated with the role. Roles can be organized hierarchically. Figure 3 illustrates an example of role hierarchy.
- *Administrative policies* regulate who can grant and revoke authorizations in the system.

Note that groups and roles are different concepts with two main differences:

- a group is a named collection of users and possibly other groups, and a role is a named collection of privileges, and possibly other roles;
- while role can sometimes be activated and deactivated directly by users at their discretion, the membership in a group cannot be deactivated.

These two concepts are not exclusive but complementary to each other. The hierarchical structure of data items, users/groups, and roles can be formally represented through a mathematical structure called *hierarchy*.

Definition 1 (Hierarchy). *A* hierarchy *is a triple* $(X, Y, \leq)$ *where:*

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

According to this definition, X represents primitive elements (e.g., a user or a file), and Y represents aggregate entities (e.g., a set of users or objects).

Given a system composed of the elements listed above, an *authorization* specifies which *authorization subjects* can execute which actions on which *authorization objects*. An authorization can then be represented as a triple (s, o, a), indicating that authorization subject s can execute action a over authorization object o.

In addition to positive authorizations, recent access control languages support also *negative authorizations*, that is, authorizations indicating that an authorization subject cannot execute a stated action on the specified authorization object. The combined use of positive and negative authorizations has the great advantage of allowing an easy management of *exceptions* in policy definition. For instance, if all users in the system but *Alice* can access a resource and we use only positive authorizations, it is necessary to specify for each subject but *Alice* a triple indicating that user u can access resource r. By contrast, with negative authorizations, we can simply state that *Alice* cannot access r, supposing, as a default policy, that everybody can access r.

To represent both positive and negative access rights, authorization triples become of the form (s, o, $\pm a$), where $+a$ indicates a positive authorization and $-a$ indicates a negative authorization.

Given a set of authorizations explicitly specified over the elements in the system, it is possible to obtain a set of derived authorizations obtained according to a hierarchy-based derivation. Some of the most common propagation policies (which include also some resolution policies for possible conflicts) are described below [26].

- *No propagation.* Authorizations are not propagated. For instance, a triple specified for a node is not propagated to its descendants. No propagation is applicable when non-leaf nodes can appear in an access request and therefore authorizations that apply to them as subject/object must be considered (as it is, for example, the case of roles).
- *No overriding.* Authorizations of a node are propagated to its descendants.
- *Most specific overrides.* Authorizations of a node are propagated to its descendants if not overridden. An authorization associated with a node n overrides a contradicting authorization[3] associated with any supernode of n for all the subnodes of n.
- *Path overrides.* Authorizations of a node are propagated to its descendants if not overridden. An authorization associated with a node n overrides a contradicting authorization associated with a supernode n' for all the subnodes of n only for the paths passing from n. The overriding has no effect on other paths.

The combined use of positive and negative authorizations brings now to the problem of how the two specifications should be treated when conflict authorizations are associated with the same node in a hierarchy. In these cases, different decision criteria could be adopted, each applicable in specific situations, corresponding to different *conflict resolution policies* that can be implemented. Examples of conflict resolution policies are the following.

[3] Authorizations (s, o, $+a$) and (s', o', $-a'$) are contradictory if $s = s'$, $o = o'$, and $a = a'$.

- *No conflict.* The presence of a conflict is considered an error.
- *Denials take precedence.* Negative authorizations take precedence.
- *Permissions take precedence.* Positive authorizations take precedence.
- *Nothing takes precedence.* Neither positive nor negative authorizations take precedence.

It may be possible that after the application of a propagation policy and a conflict resolution policy, some accesses are neither authorized nor denied (i.e., no authorization exists for them). A *decision policy* guarantees that for each subject there exists a permission or a prohibition to execute a given access. Two well known decision policies are the *closed policy* and the *open policy*. The closed policy allows an access if there exists a positive authorization for it, and denies it otherwise. The open policy denies an access if there exists a negative authorization for it, and allows it otherwise.

3 Logic-Based Access Control Languages

Several authorization models and access control mechanisms have been implemented. However, each model, and its corresponding enforcement mechanism, implements a single specified policy, which is built into the mechanism. As a consequence, although different policy choices are possible in theory, each access control system is in practice bound to a specific policy. The major drawback of this approach is that a single policy simply cannot capture all the protection requirements that may arise over time. Recent proposals have worked towards languages and models able to express, in a single framework, different access control policies, to the goal of providing a single mechanism able to enforce multiple policies. Logic-based languages, for their expressive power and formal foundations, represent a good candidate. The main advantages of using a logic-based language can be summarized as follows:

- the semantic of a logic language is clear and unambiguous;
- logic languages are very expressive and can be used to represent any kind of policy;
- logic languages are declarative and offer a better abstraction level than imperative programming languages.

The first work investigating logic languages for the specification of authorizations is the work by Woo and Lam [48]. Their proposal makes the point for the need of flexibility and extensibility in access specifications and illustrates how these advantages can be achieved by abstracting from the low level authorization triples and adopting a high level authorization language. Their language is essentially a many-sorted first-order language with a rule construct, useful to express authorization derivations and therefore model authorization implications and default decisions (e.g., closed or open policy).

In [5] the authors propose a temporal authorization model that supports periodic access authorizations and periodic rules. More precisely, deductive temporal rules with periodicity and order constraints are provided to derive new authorizations based on the presence or absence of other authorizations in specific periods of time. Another approach based on *active rules*, called *role triggers*, has been presented in [6]. The authors extend the RBAC model by adding temporal constraints on the enabling/disabling of roles.

Other logic-based access control languages support inheritance mechanisms and conflict resolution policies. The *Hierarchical Temporal Authorization Model* adopts the denials-take-precedence principle and does not distinguish between original and derived authorizations: an authorization can override another one independently from the category to which they belong. The main problem of this logic language is that it is not stratifiable. However, it supports a dynamic form of stratification that guarantees a polynomial computation time. A framework based on the C-Datalog language has also been presented. The framework is general enough to model a variety of access control models.

Although these proposals allow the expression of different kinds of authorization implications, constraints on authorizations, and access control policies, the authorization specifications may result difficult to understand and manage. Also, the trade-off between expressiveness and efficiency seems to be strongly unbalanced: the lack of restrictions on the language results in the specification of models that may not even be decidable and implementable in practice.

Starting from these observations, Jajodia et al. [26] worked on a proposal for a logic-based language that attempted to balance flexibility and expressiveness on the one side, and easy management and performance on the other. The language allows the representation of different policies and protection requirements, while at the same time providing understandable specifications, clear semantics (guaranteeing therefore the behavior of the specifications), and bearable data complexity. In the remainder of this section, we will describe this proposal in more details.

3.1 Flexible Authorization Framework

The *Flexible Authorization Framework* (FAF) [26] is a powerful and elegant logic-based framework where authorizations are specified in terms of a locally stratified rule base logic. FAF is based on an access control model that does not depend on any policy but is capable of representing any policy through the syntax of the model. In FAF, a data system to which protection must be ensured is formally defined as follows.

Definition 2 (Data System). *A* data system (DS) *is a 5-tuple* $(\mathsf{OTH}, \mathsf{UGH}, \mathsf{RH}, \mathsf{A}, \mathsf{Rel})$ *where:*

- $\mathsf{OTH} = (\mathsf{Obj}, \mathsf{T}, \leq_{\mathsf{OT}})$ *is an object hierarchy;*
- $\mathsf{UGH} = (\mathsf{U}, \mathsf{G}, \leq_{\mathsf{UG}})$ *is a user-group hierarchy;*
- $\mathsf{RH} = (\emptyset, \mathsf{R}, \leq_{\mathsf{R}})$ *is a role hierarchy;*
- A *is a set of actions;*
- Rel *is a set of relationships that can be defined on the different elements of DS;*

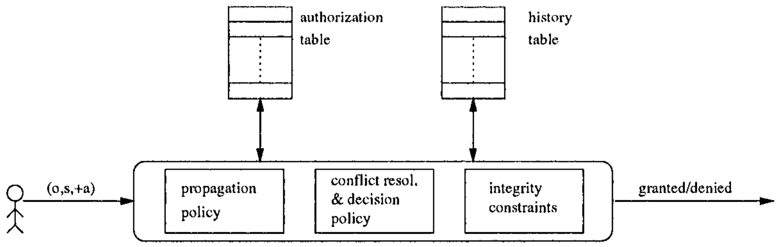


Fig. 4. Functional authorization architecture [26]

- OTH, UGH, *and* RH *are disjoint.*

Note that this definition of data system is very general as it may be used to represent any system by appropriately instantiating the five components listed above. Also, a system with no user-group, object, or role hierarchy can be represented by this definition. For instance, a data system with no role hierarchy is represented by a data system where $x \leq_{\mathsf{R}} y$ iff $x = y$.

Given an authorization triple $(s, o, \pm a)$, the authorization subject s can be a user, a group, or a role and the corresponding hierarchy, called *authorization subject hierarchy* (ASH), is intuitively obtained placing the two hierarchies UGH and RH side by side. The authorization object o can be an object, a type, or a role and the corresponding hierarchy, called *authorization object hierarchy* (AOH), is obtained placing the OTH and the inverse of RH side by side. The reason why the RH hierarchy is inverted, is to simplify the propagation rule for authorization objects: an authorization over a set of objects propagates down in the object hierarchy, while an authorization over a role propagates up in the role hierarchy. By inverting the RH hierarchy, we can simply propagate authorizations down in the authorization object hierarchy.

As depicted in Fig. 4, FAF includes the following components.

- A *history table* whose rows describe the accesses executed.
- An *authorization table* whose rows are authorizations composed of the triples (s, o, $\langle sign \rangle a$), where s is the subject, o the data item, a the action and $\langle sign \rangle$ may be '$+$' if the action is allowed and '$-$' if it is denied. This is the set of explicitly specified authorizations.
- The *propagation policy* specifies how to obtain new derived authorizations from those explicitly stored in the authorization table. Typically, derived authorizations are obtained according to hierarchy-based derivation policies. However, derivation policies are not restricted to this particular form of derivation. It is important to note that different propagation policies can be adopted in different hierarchies (ASH, AOH) and that, in the same structure, different sub-hierarchies may follow different policies.
- The *conflict resolution policy* describes how possible conflicts between the (explicit and/or derived) authorizations should be solved.

- A *decision policy* defines the response that should be returned to each access request. In case of conflicts or gaps (i.e., when an access is neither authorized nor denied), the decision policy determines the answer. In many systems, decisions assume either the open or the closed policy, where, by default, access is granted or denied, respectively.
- A set of *integrity constraints* that may impose restrictions on the content and output of the other components. Integrity rules can be used to individuate errors in the hierarchies or in the explicitly specified authorizations, or for implementing duty separation.

When a subject s requires the execution of action a on object o, the system needs to verify whether the authorization (s, o, $+a$) or (s, o, $-a$) can be derived using the authorization table, propagation policy, history table, conflict resolution policy, and decision policy that have been defined in the system. If a positive authorization is derived, then the access is allowed. Otherwise, if a negative authorization is derived, the access is denied.

As previously discussed, FAF allows the representation of different propagation policies, conflict resolution policies, and decision policies that a security system officer (SSO) might want to use. However, these policies represent only some of the possibilities and FAF is flexible enough to allow a SSO to express what she needs for her applications. To address this issue, the functional authorization architecture can be realized through the following approach:

- the authorization table is viewed as a database;
- policies are expressed by a restricted class of logic programs, called *authorization specification*, which have certain properties;
- the semantics of authorization specifications is given through the well known stable model semantics and well founded model semantics of logic programs, ensuring thus the existence of exactly one stable model;
- accesses will be allowed or denied on the basis of the truth value of an atom associated with the access in the unique stable model.

Accordingly, the *authorization specification logic* language (ASL) is a logic language used to encode the system security needs. ASL is created from the following alphabet.

- *Constant symbols* are members of the sets of users U, groups of users G, objects Obj, types of objects T, roles R, and actions A.
- *Variable symbols* are variables ranging over the sets U, G, Obj, T, R, and A.
- *Predicate symbols* are partitioned into three categories. The first category contains predicates needed to express the *access control policy*:
  - `cando`(o, s, $\pm a$) explicitly represents the authorizations defined by the SSO: it allows (or denies, depending on the sign) subject s to execute action a on object o;
  - `dercando`(o, s, $\pm a$) represents authorizations derived by the system;

- do(o, s, $\pm a$) represents the accesses that must be allowed or denied, and are obtained after the application of the conflict resolution and decision policies;
- done(o, s, r, a, t) keeps the history of the accesses executed. A fact of the form done(o, s, r, a, t) indicates that s operating in role r executed action a on object o at time t.
- over represents overriding policies in the authorization subject and/or authorization object hierarchies;
- error signals errors in the specification or use of authorizations; it can be used to enforce static and dynamic constraints on the specifications.

The second category of predicate symbols is the hie-predicates for the evaluation of hierarchical relationships between the elements of the data system (e.g., user's membership in groups, inclusion relationships between objects). There are two predicates:

- in(x, y, H) evaluates to true only if $x \leq y$ in the structure represented by hierarchy H, where H is ASH or AOH;
- dirin(x, y, H) evaluates to true only if x is a direct descendant of y in the hierarchy H, where H is ASH or AOH.

The third category of predicates is the rel-predicates that are used to express different relationships between elements in the data system. These predicates are not fixed by the model and are application specific. Examples of such predicates are the following:

- owner(o, s) specifies that subject s is the owner of object o in the system;
- isuser(s), isgroup(g), and isrole(r) evaluate to true only if their argument is a user, a group, or a role, respectively.

If p is one of the above-mentioned predicate symbols with arity n and $t_1, \ldots, t_n$ are terms appropriate for p, then $p(t_1, \ldots, t_n)$ is an *atom*. A *literal* is an atom or its negation. All these predicates, atoms and literals can be exploited to express a policy. We now illustrate how each component in Fig. 4 is represented by a set of rules.

History table. It contains only done predicates to keep track of the past accesses performed by the users. The instances of the done predicate are stored in a relation table with schema (Object, User, Role, Action, Time). For instance, done(*IS01*, *David*, *Admin*, *read*, *15/05/2005 15:30*) denotes a *read* on object *IS01* executed by *David* playing role *Admin* at time *15/05/2005 15:30*.

Authorization table. It contains a finite set of *authorization rules* of the form:

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

where o is an object or an object type, s is a user or a group, $\langle sign \rangle$ is either '+' or '−', a is an action, and $L_1, \ldots, L_n$ are either done, hie- or rel- literals. If these literals are evaluated to true, the authorization on the left of the rule is granted. For instance, rule cando(*IS02*, s, $+r$) $\leftarrow$ in(s, *Sales*, ASH) & $\neg$done(*IS02*,s, w, t) states that members of group *Sales* can read object *IS02* if they have not already modified object *IS02* at time t.

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

Fig. 5. Rules enforcing different propagation policies on ASH

Propagation policies. The propagation policy is composed of two sets of rules: *overriding rules*, stating when an authorization can override another one; and *derivation rules*, representing the set of authorizations that can be derived by the authorizations explicitly defined. Overriding rules can be defined on the authorization subject hierarchy ($\texttt{over}_{\mathsf{AS}}$) or on the authorization object hierarchy ($\texttt{over}_{\mathsf{AO}}$) and are rules of the form:

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

where o and o' are objects or object types, s and s' are users or groups, $\langle sign\rangle$ is either '+' or '$-$', a is an action, and $L_1, \ldots, L_n$ are either cando, done, hie-, or rel- literals. If these literals evaluate to true, the overriding rule is applied. The derivation rules are of the form:

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

where o is an object or an object type, s is a user or a group, $\langle sign\rangle$ is either '+' or '$-$', a is an action, and $L_1, \ldots, L_n$ are either cando, over, dercando, done, hie-, or rel- literals. If these literals evaluate to true, the derivation rule is applied. For instance, rule dercando(*Received*, s, $+r$) $\leftarrow$ dercando(*Orders*, s, $+r$) derives a permission for a subject to read object type *Received* if there exists an (explicit or implicit) authorization for the subject to read object type *Orders*.

The set of dercando rules in the system is composed of all the authorizations that can be derived through the propagation policy defined by the SSO. Figure 5 illustrates the set of rules enforcing the most common propagation policies on the ASH hierarchy.

Conflict resolution and decision policies. The conflict resolution and decision policies allow the SSO to specify how conflicts are to be solved. A *decision rule* is a rule of the form:

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

where o is an object or an object type, s is a user or a group, a is an action, and $L_1, \dots, L_n$ are either `cando`, `dercando`, `done`, `hie`- or `rel`- literals. In addition to these positive decision rules, there is also the rule: $\texttt{do}(o,\ s,\ -a) \leftarrow \neg\texttt{do}(o,\ s,\ +a)$. This rule guarantees the completeness of the policy, that is, for each triple $(o,\ s,\ a)$, one of the two $\texttt{do}(o,\ s,\ +a)$ or $\texttt{do}(o,\ s,\ -a)$ holds. Intuitively, the set of atoms $\texttt{do}(o,\ s,\ +a)$ specifies the set of all authorized accesses. For instance, rule $\texttt{do}(o,\ s,\ +r) \leftarrow \neg\texttt{dercando}(o,\ s,\ +r)\ \&\ \neg\texttt{dercando}(o,\ s,\ -r)\ \&\ \texttt{in}(o,\ \mathit{Invoices},\ \text{AOH})$ states that a subject s can read an object o if no authorization has been derived for s on an object of type *Invoices*. Figure 6 illustrates possible rules enforcing the most common conflict resolution and decision policies.

Integrity rules. Since there is a great potential for errors in the authorization specifications, it is possible to specify integrity rules defining constraints that must hold on the authorization specifications. An *integrity rule* is a rule of the form:

$$\texttt{error} \leftarrow L_1 \& \dots \& L_n$$

where $L_1, \dots, L_n$ are either `cando`, `dercando`, `done`, `do`, `hie`- or `rel`- literals. If these literals evaluate to true, an error occurs. Restrictions imposed through integrity constraints can be both general or specific to an application. For instance, rule $\texttt{error} \leftarrow \texttt{cando}(o,\ s,\ +a)\ \&\ \texttt{cando}(o,\ s,\ -a)$ states that an error occurs if there are two contradictory explicit `cando` predicates.
The integrity rules are evaluated after the access decision has been taken and can block its execution if an error is derived. They are also checked whenever a change occurs in some table used by the authorization framework: if the change implies an error, the corresponding operation is denied.

Authorization specifications are stated as logic rules defined on the predicates of the language. To ensure clean semantics and implementability, the format of the rules is restricted to guarantee (local) stratification of the resulting program (see Fig. 7).[4] The stratification also reflects the different semantics given to the predicates: `cando` will be used to specify basic authorizations, `dercando` will be used to enforce implication relationships and produce derived authorizations, and `do` to take the final access decision. Stratification ensures that the logic program corresponding to the rules has a unique stable model, which coincides with the well founded semantics [22]. Also, this model can be effectively computed in polynomial time. The authors of FAF also present a materialization technique for producing and storing the model corresponding to a set of logical rules. The model is computed on the initial specifications and updated with incremental maintenance strategies.

[4] A program is locally stratified if there is no recursion among predicates going through negation.

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

Fig. 6. Conflict resolution and decision policies rules

Stratum	*Predicate*	*Rules defining predicates*
0	hie-predicates	Base relations.
	rel-predicates	Base relations.
	done	Base relation.
1	cando	Body may contain done, hie-, and rel- literals.
2	dercando	Body may contain cando, dercando, done, hie-, and rel- literals. Occurrences of dercando literals must be positive.
3	do	When head is of the form do(_, _, $+a$), body may contain cando, dercando, done, hie-, and rel- literals.
4	do	When head is of the form do(o, s, $-a$), body contains just one literal ¬do(o, s, $+a$).
5	error	Body may contain do, cando, dercando, done, hie-, and rel- literals.

Fig. 7. Rule composition and stratification of FAF

An Example of FAF Application

A simplified scenario is constructed to describe the application of the FAF model and language. Consider an online computer store where objects are organized according to the hierarchy in Fig. 2, and users are grouped as illustrated in Fig. 1. Suppose also that the system does not use roles.

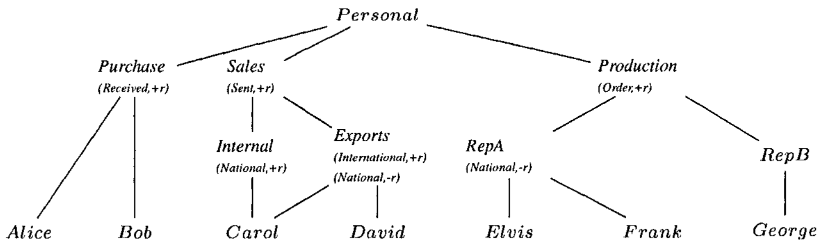


Fig. 8. An example of labeled user-group hierarchy

The SSO defines the set of initial `done`, `hie-`, and `rel-` literals. For simplicity, we assume that the system has no `done` predicates. The `dirin`-literals necessary for the definition of the subject and object hierarchies follow the arcs in the graphs in Fig. 1 and in Fig. 2, respectively.

From the `dirin` literals explicitly specified by the SSO, it is possible to verify the validity of the `in` literals. The following are examples of protection requirements, where *r* is used to denote the read action.

- Members of the *Purchase* group can read *Received Invoices*.
  cando(o, s, $+r$) $\leftarrow$ in(s, *Purchase*, ASH) & in(o, *Received*, AOH)
- Members of the *Sales* group can read *Sent Invoices*.
  cando(o, s, $+r$) $\leftarrow$ in(s, *Sales*, ASH) & in(o, *Sent*, AOH).
- Members of the *Internal* group can read *National Orders*.
  cando(o, s, $+r$) $\leftarrow$ in(s, *Internal*, ASH) & in(o, *National*, AOH).
- Members of the *Exports* group can read *International Orders*.
  cando(o, s, $+r$) $\leftarrow$ in(s, *Exports*, ASH) & in(o, *International*, AOH).
- Members of the *Production* group can read any kind of *Orders*.
  cando(o, s, $+r$) $\leftarrow$ in(s, *Production*, ASH) & in(o, *Orders*, AOH).
- Members of the *RepA* group cannot read *National Orders*.
  cando(o, s, $-r$) $\leftarrow$ in(s, *RepA*, ASH) & in(o, *National*, AOH).
- Members of the *Exports* group cannot read *National Orders*.
  cando(o, s, $-r$) $\leftarrow$ in(s, *Exports*, ASH) & in(o, *National*, AOH).

After the definition of these explicit authorizations, the SSO needs to choose a propagation policy. Suppose that the *most specific overrides* principle has been chosen and that the propagation is performed on the authorization subject hierarchy ASH. First, for each explicit authorization (s, o, $\pm a$), the propagation process associates with subject s in the hierarchy a pair of the form $\langle obj, \pm a\rangle$. Figure 8 illustrates the resulting labeled hierarchy.

The authorizations are then propagated along the hierarchy thus obtaining the following set of `dercando` literals.

- dercando(*Received*, *Purchase*, $+r$)
- dercando(*Received*, *Alice*, $+r$)
- dercando(*Received*, *Bob*, $+r$)
- dercando(*Sent*, *Sales*, $+r$)

- `dercando`(*Sent*, *Internal*, $+r$); `dercando`(*National*, *Internal*, $+r$)
- `dercando`(*Sent*, *Exports*, $+r$); `dercando`(*International*, *Exports*, $+r$); `dercando`(*National*, *Exports*, $-r$)
- `dercando`(*Sent*, *David*, $+r$); `dercando`(*International*, *David*, $+r$); `dercando`(*National*, *David*, $-r$)
- `dercando`(*Sent*, *Carol*, $+r$); `dercando`(*International*, *Carol*, $+r$); `dercando`(*National*, *Carol*, $+r$); `dercando`(*National*, *Carol*, $-r$)[5]
- `dercando`(*Orders*, *Production*, $+r$)
- `dercando`(*International*, *RepA*, $+r$); `dercando`(*National*, *RepA*, $-r$)
- `dercando`(*International*, *Elvis*, $+r$); `dercando`(*National*, *Elvis*, $-r$)
- `dercando`(*International*, *Frank*, $+r$); `dercando`(*National*, *Frank*, $-r$)
- `dercando`(*Orders*, *RepB*, $+r$)
- `dercando`(*Orders*, *George*, $+r$)

It is easy to see that there are some conflicts. The first conflict arises because members of the *RepA* group can read the objects in *Orders* and cannot read objects in the subset *National*. However, according to the *most specific overrides* principle, members of the group *RepA* can read *Orders* that do not belong to the *National* category and cannot read objects in the *National* category.

The second conflict involves user *Carol* who is a member of the group *Internal* and group *Exports* and for which there is a positive and negative authorization on *National*, respectively. In this case, the conflict can be solved by applying, for example, the *denials take precedence principle* together with the *closed* policy. The result of this last step is a set of `do` literals representing all the triples (s, o, $\pm a$) derivable in the structure.

We now examine some examples of access requests and analyze whether these requests will be granted or denied.

Request 1. *Alice* requests to read object *IS02*.
Access denied. There is neither an explicit authorization nor an implicit authorization and therefore, according to the default closed policy, the access is denied.

Request 2. *Carol* requests to read object *ON01*.
Access denied. Object *ON01* is a member of class *National* and according to the denials take precedence principle *Carol* cannot read national orders.

Request 3. *Frank* requests to read object *ON01*.
Access denied. Frank is a member of group *RepA*, object *ON01* is a member of class *National* and, according to the most specific overrides principle, the *RepA* group cannot read national orders.

Request 4. *Frank* requests to read object *OI01*.
Access allowed. Frank is a member of group *RepA* and object *OI01* is a member of class *Orders* and is not a member of class *National*.

[5] There is a conflict that cannot be solved at this point of the policy evaluation process.

4 XML-Based Access Control Languages

Although logic-based access control models and languages are powerful and expressive, they are not immediately suited to the Internet context, where simplicity and easy integration with existing technology must be ensured. Therefore, an interesting aspect to be addressed concerns the definition of a language for expressing and exchanging policies based on a high level formulation that, while powerful, can be easily interchangeable and both human and machine readable. Insights in this respect can be taken from recent proposals expressing access control policies as XML documents. Indeed, the *eXtensible Markup Language* (XML) [49], a markup metalanguage standardized by the World Wide Web Consortium (W3C), is the standard language for information exchange on the Internet and many XML-based access control languages have been proposed. The first advantage of this class of access control languages is their simplicity in policy definition. Another important advantage of XML-based access control languages is the *interoperability*, that consists in the possibility of exchanging policies through different systems using the same access control language. This feature is particularly interesting in an open environment like the Internet, where a single system, which has to be protected as a single entity, may be distributed over the Net.

Initially, XML-based access control languages were thought only for the protection of resources that were themselves XML files [14, 15, 20, 21]. In [14, 15] authorizations can be positive and negative and can be defined both at the document-level or at the Document Type Definition (DTD) level (in this case authorizations propagate to all instances of the DTD). Authorizations are characterized by a type field defining how the authorizations must be treated with respect to propagation at finer granules and overriding (exception support). The model in [29] supports read and write privileges. The authors define three types of propagation policies: no propagation, propagation up (an authorization referring to an element is propagated to all its parent elements) or propagation down (an authorization referring to an element is propagated to all its sub-elements). The conflict resolution policy is either "denials take precedence" or "permissions take precedence". The main contribution of this paper is the definition of *provisional authorizations* that specify actions that a user has to perform before obtaining a given privilege. The model in [21] supports the read privilege only. The authors do not define any propagation policy. The conflict resolution policy is based on the priority of the different rules. More recently, in [20] has been proposed an approach that tries to address the `write` privilege based on the non-standard XML update language `XUpdate`. The author separates the existence of an XML value and its content adding a new position privilege that allows to know the existence of a node but not its content. Nodes tagged with a position privilege are shown with a `restricted` label.

These proposals have the common characteristic that they present a model for securing XML documents. Recent proposals instead use XML to define languages for expressing protection requirements on any kind of data/resources [2, 13, 17, 39]. Two relevant XML-based access control languages are WS-Policy [13] and the *eXtensible Access Control Markup Language* (XACML) [17]. Based on the WS-

Security [3], WS-Policy provides a grammar for expressing Web service policies. The WS-Policy includes a set of general messaging related assertions defined in WS-PolicyAssertions [11] and a set of security policy assertions related to supporting the WS-Security specification defined in WS-SecurityPolicy [44]. In addition, WS-PolicyAttachment [12] defines how to attach these policies to Web services or other subjects such as service locators. XACML is the result of a recent OASIS standardization effort proposing an XML-based language to express and interchange access control policies. XACML is designed to express authorization policies in XML against objects that can themselves be identified in XML. The XACML language has the great advantage that it can be used to express a variety of different policies and has the basic functionalities of most policy representation mechanisms. Moreover, XACML has standard *extension points* for defining new functions, data types, combining logic, and so on. While XACML and WS-Policy share some common characteristics, XACML has the advantage of enjoying an underlying policy model as a basis, resulting in a clean and unambiguous semantics of the language [2]. For this reason, XACML is the most common XML-based access control language used. In the remainder of this Section, we illustrate XACML as our choice of language.

4.1 XACML Policy Definition

XACML relies on a model that provides a *formal* representation of the access control security policy and its working. This modeling phase is essential to ensure a clear and unambiguous language which could otherwise be subject to different interpretations and uses. The main concepts of interest in the XACML policy language model are *rule*, *policy*, and *policy set*.

An XACML policy has, as root element, either a `Policy` or a `PolicySet`. A `PolicySet` is a collection of `Policy` or `PolicySet`. An XACML policy consists of a set of *rules*, a *target*, an optional set of *obligations*, and a *rule combining algorithm*. A `Rule` specifies a permission (`permit`) or a denial (`deny`) for a subject to perform an action on an object. A `Target` basically consists of a simplified set of conditions for the subject, resource, and action that must be satisfied for a policy to apply to a given request. If all the conditions of a `Target` are satisfied, then its associated `Policy` (or `PolicySet`) applies to the request. An `Obligation` is an operation that has to be performed in conjunction with the enforcement of an authorization decision. Each `Policy` also defines a rule combining algorithm used for reconciling the decisions each rule makes. The final decision value, called *authorization decision*, is the value of the policy as defined by the rule combining algorithm. An authorization decision can be `permit`, `deny`, `not applicable` (when no applicable policies or rules could be found), or `indeterminate` (when some errors occurred during the access control process). XACML defines different combining algorithms such as *deny overrides* (i.e., denials take precedence), *permit overrides* (i.e., permissions take precedence), *first applicable* (i.e., the first applicable rule is considered), and *only-one-applicable* (i.e., a deny or permit result is obtained only if exactly one rule is applicable).

The `PolicySet` element is similar to the `Policy` element and consists of a set of *policies* (instead of *rules*), a *target*, an optional set of *obligations*, and a *policy combining algorithm* (instead of a rule combining algorithm).

The `Rule` element specifies the actual conditions under which access is to be allowed or denied. The components of a rule are an optional *target*, an *effect*, and a *condition*. The target defines the set of resources, subjects, and actions to which the rule is intended to apply. The effect of the rule can be `permit` or `deny`. The condition represents a boolean expression that may further refine the applicability of the rule.

An important feature of XACML is that a rule is based on the definition and evaluation of attributes corresponding to specific characteristics of a subject, resource, action, or environment. Any request is mainly composed of attributes that will be compared to attribute values in a policy to make an access decision. Attributes are identified by the `SubjectAttributeDesignator`, `ResourceAttributeDesignator`, `ActionAttributeDesignator`, and `EnvironmentAttributeDesignator` elements. These elements use the `AttributeValue` element to define the value of a particular attribute. Alternatively, the `AttributeSelector` element can be used to specify where to retrieve a particular attribute. Note that both the attribute designator and attribute selector elements can return multiple values. To this reason, XACML provides an attribute type, called *bag*, that is an unordered collection and can contain duplicate values for a particular attribute. To correctly handle the data type bag, XACML has a powerful set of functions that can work on arbitrary collections of values and return any kind of attribute value supported in the system. Functions can also be nested, that is, the output of a function is the input of another. The XACML defines a set of basic functions that can be enriched by adding application-specific functions. Since often resources are represented in a hierarchical structure (e.g., file system), XACML v. 2.0 introduces a method for handling hierarchical resources (see Sect. 2). More precisely, XACML v. 2.0 provides a mechanism for:

- representing the identity of a node;
- requesting access to a node;
- stating policies that apply to one or more nodes.

The hierarchy can be both a tree or a forest and cannot have cycles. It is important to note that there are two different ways for representing a resource in a hierarchy [42]. In the first one, the hierarchy to which the node belongs is represented as a XML document and the resource is represented as a node in the XML document. In the second case, the hierarchy is not represented as a XML document and has no representation. Analogously, subjects can be hierarchically represented (see Sect. 2) but XACML does not offer any functionality for managing groups of subjects. This is mainly due to the fact that XACML is used in distributed systems, consequently the resource handler cannot know the whole user-group hierarchy. However, XACML provides a way for checking at any time if a user belongs to a specific group: when a request for a resource is submitted, the resource handler checks the requester's properties, as these are automatically inserted in the same request. Among these properties, there is the set of groups to which the user belongs. XACML also supports the role-based access control [19].

As a simple example of policy, consider the example introduced in Sect. 3.1. Suppose that the online computer store defines the following high level policy: "Members of the *Sales* group can read invoice *IS02*".

Figure 9 shows the XACML policy corresponding to this high level policy. The policy applies to requests on the `http://www.example.com/documents/invoices/sent/IS02.xml` resource. It has one rule with a target that requires a read action and a condition that evaluates to true only if the subject is a member of the group *Sales*.

4.2 XACML Request and Response

XACML defines also a standard format for expressing requests and responses. Each request contains attributes for the subject, resource, action, and, optionally, for the environment. More precisely, each request includes exactly one set of attributes for the resource and action and at most one set of environment attributes. There may be multiple sets of subject attributes each of which is identified by a category URI.

A response element contains one or more results each of which corresponds to the result of an evaluation. Each result contains three elements, namely `Decision`, `Status`, and `Obligations`. The `Decision` element specifies the authorization decision, the `Status` element indicates if some error occurred during the evaluation process, and the optional `Obligations` element states the obligations to fulfill.

As an example, suppose that *Carol* wants to read the `http://www.example.com/documents/invoices/sent/IS02.xml` resource. Figure 10 illustrates the corresponding XACML request. This request is compared with the XACML policy in Fig. 9. The result is that the user is allowed to access the requested resource.

```
<Policy PolicyId="SentInvoice" RuleCombiningAlgId="urn:oasis:names:tc:
xacml:1.0:rule-combining-algorithm:deny-overrides">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:
        function:anyURI-equal">
          <AttributeValue
          DataType="http://www.w3.org/2001/XMLSchema#anyURI">
            http://www.example.com/documents/invoices/sent/IS02.xml
          </AttributeValue>
          <ResourceAttributeDesignator
          DataType="http://www.w3.org/2001/XMLSchema#anyURI"
          AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
        </ResourceMatch>
      </Resource>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>
  <Rule RuleId="ReadRule" Effect="Permit">
    <Target>
      <Subjects>
        <AnySubject/>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:
          function:string-equal">
            <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string">
              read
            </AttributeValue>
            <ActionAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:
    function:string-equal">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:
      function:string-one-and-only">
        <SubjectAttributeDesignator
        DataType="http://www.w3.org/2001/XMLSchema#string"
        AttributeId="group"/>
      </Apply>
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
        Sales
      </AttributeValue>
    </Condition>
  </Rule>
</Policy>
```

Fig. 9. An example of XACML policy

```
<Request>
  <Subject>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
      DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name">
        <AttributeValue>Carol@example.com</AttributeValue>
    </Attribute>
    <Attribute AttributeId="group"
      DataType="http://www.w3.org/2001/XMLSchema#string"
      Issuer="administrator@example.com">
        <AttributeValue>Sales</AttributeValue>
    </Attribute>
  </Subject>
  <Resource>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
      DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      <AttributeValue>http://www.example.com/documents/invoices/sent/IS02.xml
      </AttributeValue>
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
      DataType="http://www.w3.org/2001/XMLSchema#string">
        <AttributeValue>read</AttributeValue>
    </Attribute>
  </Action>
</Request>
```

Fig. 10. An example of XACML request

4.3 XACML Architecture

Figure 11 illustrates the main entities involved in the XACML domain. The standard gives a definition of these concepts that we summarize as follows.

- The *Policy Evaluation Point* (PEP) module receives initially the access request in a naive format and passes it to the Context Handler. Similarly, when a decision has been taken by the decision point, PEP enforces the access decision that it receives from the Context Handler.
- The *Policy Decision Point* (PDP) module receives an access request and interacts with the PAP that encapsulates the information needed to identify the applicable policies. It then evaluates the request against the applicable policies and returns the authorization decision to the Context Handler module.
- The *Policy Administration Point* (PAP) module is an interface for searching policies. It retrieves the policies applicable to a given access request and returns them to the PDP module.
- The *Policy Information Point* (PIP) module provides attribute values about the subject, resource, and action. It interacts directly with the Context Handler.
- The *Context Handler* translates the access requests in a native format into a canonical format. Basically, it acts as a bridge between PDP and PEP modules and it is in charge for retrieving attribute values needed for policy evaluation.
- The *Environment* provides a set of attributes that are relevant to take an authorization decision and are independent from a particular subject, resource, and action.

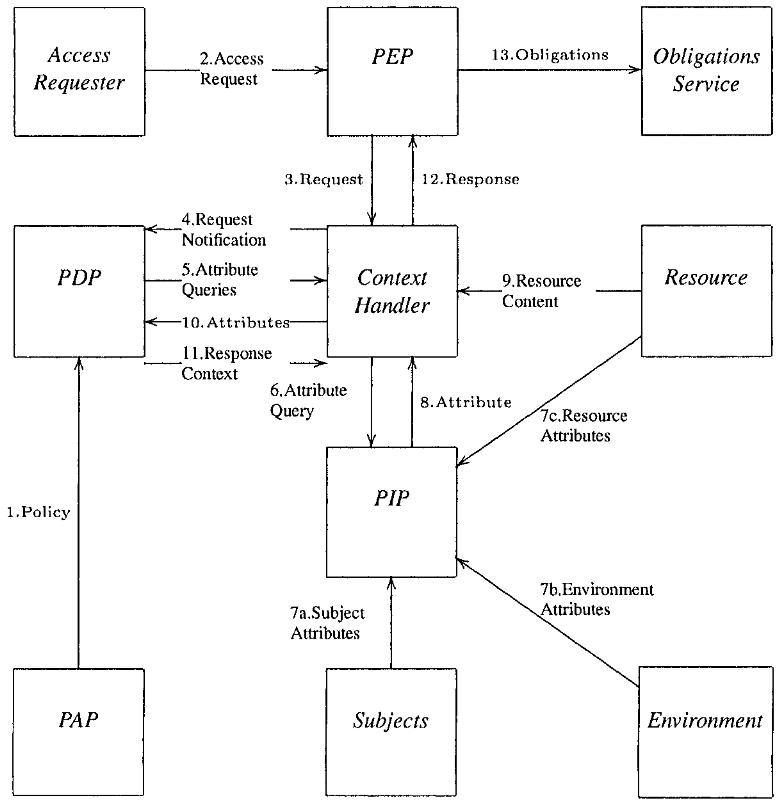


Fig. 11. XACML overview [17]

- The *Obligations Service* module manages obligations, which are the operations that should be performed by the PEP when enforcing the final authorization decision.
- The *Access Requester* module makes requests to the system in a naive form.
- A *Resource* is a service or a data collection available for requests.
- The *Subjects* are the actors of the system; they usually have attributes that can be used in predicates.

The XACML data-flow (Fig. 11) is not limited to the phase of evaluating an access request but involves also an initialization phase. More precisely, the data-flow consists of the following steps.

1. The policies are made available by the PAP to the PDP to fulfill the resource owner needs.
2. The access requester communicates her request to the PEP module in a naive format.

3. The PEP transmits the original request to the Context Handler, possibly together with attributes of the subject, resource, action and environment involved in the request.
4. The Context Handler builds *XACML request context*, with the information provided by the PEP and sends it to the PDP.
5. In case of additional attributes of the subject, resource, action, or environment are needed, the PDP asks for them to the Context Handler.
6. The Context Handler sends the attribute request coming from the PDP to the PIP module.
7. The PIP module retrieves the attributes interacting directly with Subject, Resource, and Environment.
8. The PIP sends the attributes just obtained to the Context Handler.
9. The Context Handler inserts the resource in the context created at step 4.
10. The attributes obtained from the PIP and eventually the resource involved in the access request are sent by the Context Handler to the PDP. The PDP can now evaluate the policies and take a decision.
11. The PDP sends to the Context Handler the *XACML response context* that includes the final decision.
12. The Context Handler translates the *XACML response context* in the naive format used by the PEP module and sends the final response to the PEP.
13. The PEP fulfills the obligations included in the response and, if the access is permitted, the PEP grants access to the resource. Otherwise, the access is denied.

Although XACML is suitable for a variety of different applications, the PDP module needs to receive standardized input and returns standardized output. Therefore, any implementation of XACML has to be able to translate the attribute representation in the application environment (e.g., SAML or CORBA) in the corresponding XACML context.

5 Credential-Based Access Control Languages

Open environments are characterized by a number of systems offering different services. In such a scenario, interoperability is a very important issue and traditional assumptions for establishing and enforcing access control regulations do not hold anymore. A server may receive requests not just from the local community of users, but also from remote, previously unknown users. The server may not be able to authenticate these users or to specify authorizations for them (with respect to their identity). The traditional separation between *authentication* and *access control* cannot be applied in this context, and alternative access control solutions should be devised. A possible solution to this problem is represented by the use of *digital certificates* (or *credentials*), representing statements certified by given entities (e.g., certification authorities), which can be used to establish properties of their holder (such as identity, accreditation, or authorizations) [18, 23].

The development of access control systems based on credentials is not a simple task and the following issues need to be investigated [10].

- *Ontologies.* Due to the openness of the scenario and the richness and variety of security requirements and attributes that may need to be considered, it is important to provide parties with a means to understand each other with respect to the properties they enjoy (or request the counterpart to enjoy). Therefore, common languages, dictionaries, and ontologies must be developed.
- *Client-side and server-side restrictions.* In an open scenario, *mutual access control* is an important security feature in which a client should be able to prove its eligibility for a service, and the server communicates to the client the requirements it needs to satisfy to get access.
- *Credential-based access control rules.* It is necessary to develop languages supporting access control rules based on credentials and these languages have to be flexible and expressive enough for users. The most important challenge in defining a language is the trade off between expressiveness and simplicity: the language should be expressive enough for defining different kinds of policies and simple enough for the final user.
- *Access control evaluation and outcome.* Users may be occasional and they may not know under what conditions a service can be accessed. Therefore, to make a service "usable", access control mechanisms cannot simply return "yes" or "no" answers. It may be necessary to explain why accesses are denied, or - better - how to obtain the desired permissions. Therefore, the system can return an *undefined response* meaning that current information is insufficient to determine whether the request can be granted or denied. For instance, suppose that a user can access a service if she is at least eighteen and can provide a credit card number. Two cases can occur: *i)* the system knows that the user is not yet eighteen and therefore returns a negative response; *ii)* the user has proved that she is eighteen and the system returns an undefined response together with the request to provide the number of a credit card.
- *Privacy-enhanced policy communication.* Since the server does not return a simple yes/no answer to access requests, but returns the set of credentials that clients have to submit for obtaining access, there is a need for correctly and concisely representing them. The naive way to formulate a credential request, that is, giving the client a list with all the possible sets of credentials that would enable the service, is not feasible, due to the large number of possible alternatives. Also, the communication process should not disclose "too much" of the underlying security policy, which might also be regarded as sensitive information. Analogously, the client should be able to select in private a minimal set of credentials whose submission will authorize the desired service.

Blaze et al. [8] presented an approach for accessing services on the Web. This work is therefore limited to the Web scenario and is based on identity certificates only. The first proposals investigating the application of credential-based access control regulating access to a server were made by Winslett et al. [38, 47]. Here, access control rules are expressed in a logic language and rules applicable to an access can be communicated by the server to clients. In [46, 50] the authors investigate trust negotiation issues and strategies that a party can apply to select credentials to submit to the opponent party in a negotiation. More recently, in [50] the *PRUdent NEgotiation Strategy* (PRUNES) has been presented. This strategy ensures that the client communicates its credentials to the server only if the access will be granted and the set of certificates communicated to the server is the minimal necessary for granting it. Each party defines a set of *credential policies* that regulates how and under what conditions the party releases its credentials. The negotiation consists of a series of requests for credentials and counter-requests on the basis of the parties' credential policies. The credential policies established can be graphically represented through a tree, called *negotiation search tree*, composed of two kinds of nodes: *credential nodes*, representing the need for a specific credential, and *disjunctive nodes*, representing the logic operators connecting the conditions for credential release. The root of a tree node is a service (i.e., the resource the client wants to access). The negotiation can therefore be seen as a backtracking operation on the tree. The backtracking can be executed according to different strategies. For instance, a *brute-force* backtracking is complete and correct, but is too expensive to be used in a real scenario. The authors therefore propose the PRUNES method that prunes the search tree without compromising completeness or correctness of the negotiation process. The basic idea is that if a credential C has just been evaluated and the state of the system is not changed too much, than it is useless to evaluate again the same credential, as the result will be exactly as the result previously computed.

It has been demonstrated that the PRUNES method is correct and that the communication time is $O(n^2)$ and the computational time is $O(n \cdot m)$, where n is the number of credentials involved in the trust establishment process, and m is the total size of the credential disclosure policies related to the same credentials.

The same research group proposed also a method for allowing parties adopting different negotiation strategies to interoperate through the definition of a *Disclosure Tree Strategy* (DTS) family [52]. The authors show that if two parties use different strategies from the DST family, they are able to establish a negotiation process. The DTS family is a closed set, that is, if a negotiation strategy can interoperate with any DST strategy, it must also be a member of the DST family.

In [51] a *Unified Schema for Resource Protection* (UniPro) has been proposed. This mechanism is used to protect the information in policies. UniPro gives (opaque) names to policies and allows any named policy P_1 to has its own policy P_2 meaning that the contents of P_1 can only be disclosed to parties who have shown that they satisfy P_2. Another approach for implementing access control based on credentials is the *Adaptive Trust Negotiation and Access Control* (ATNAC) [36]. This method grants or denies access to a resource on the basis of a *suspicion level* associated with subjects. The suspicion level is not fixed but may vary on the basis of the probability that the user has malicious intents. In [43] the authors propose to apply the automated trust negotiation technology for enabling secure transactions between portable devices that have no pre-existing relationship.

In [53] the same research group proposed a negotiation architecture, called *TrustBuilder*, that is independent from the language used for policy definition and from the strategies adopted by the two parties for policy enforcement.

Other logic-based access control languages based on credentials have been introduced. For instance, D1LP and RT [32–34], the SD3 language [28], and Binder [16]. In [27, 48] logic languages are adopted to specify access restrictions in a certificate-based access control model.

5.1 A Credential-Based Access Control Language

A first attempt to provide a uniform framework for attribute-based access control specification and enforcement was presented by Bonatti and Samarati in [10]. Like in previous proposals, access regulations are specified as logical rules, where some predicates are explicitly identified. Each party has a *portfolio*, that is, a collection of credentials and declarations (unsigned statements), and has associated a set of services that can provide. Credentials are essentially digital certificates, and must be unforgeable and verifiable through the issuing certificate authority's public key; declarations are instead statements made by the user herself, that autonomously issues a declaration. Abstractions can be defined on services, grouping them in sets, called *classes*. The main advantage of this proposal is that it allows to exchange the minimal set of certificates, that is, client communicates the minimal set of certificates to the server, and the server releases the minimal set of conditions required for granting access. To this purpose, the server defines a set of *service accessibility rules*, expressing the necessary and sufficient conditions for granting access to a resource. On the other hand, both clients and severs can specify a set of *portfolio disclosure rules*, used to establish the conditions under which credentials and declarations may be released.

The rules both in the service accessibility and portfolio disclosure sets are defined through a logic language. The language includes a set of predicates whose meaning is expressed on the basis of the current *state*. The state indicates the parties' characteristics and the status of the current negotiation process, that is, the certificates already exchanged, the requests made by the two parties, and so on. The basic predicates of the language are the following.

- `credential(c, K)` evaluates to true if the current state contains certificate `c` verifiable using key `K`.
- `declaration(d)` evaluates to true if the current state contains declaration `d`, where `d` is of the form *attribute-name=value-term.*
- `cert-authority(CA,` K_{CA}`)` evaluates to true if the party using it in her policy trusts certificates issued by certificate authority `CA`, whose public key is K_{CA}.
- A set of non predefined predicates necessary for evaluating the current *state* values; these predicates can evaluate both the *persistent* and the *negotiation state*. The persistent state contains information that is stored on the site and is not related to a single negotiation but to the party itself. The negotiation state is related to the information on a single negotiation and is removed at the end of the same.

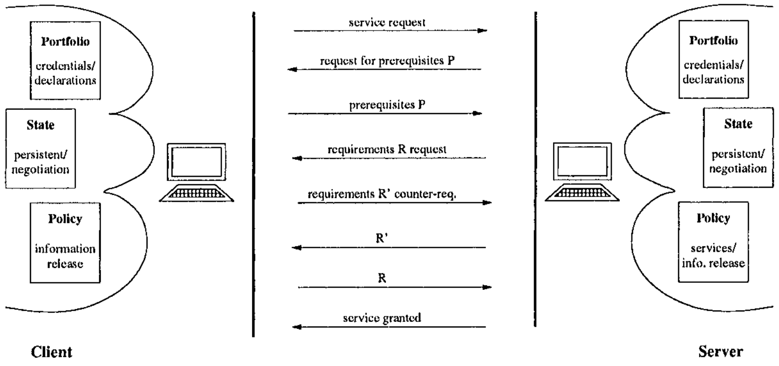


Fig. 12. Client-server negotiation

- A set of non predefined *abbreviation* predicates that are used to abbreviate requirements in the negotiation phase.
- A standard set of mathematical built-in predicates, such as $=$, $\neq$, and $\leq$.

The rules for service accessibility and portfolio disclosure have, in their body, a composition of the above-mentioned predicates, and in their head the specification of the services accessible or the certificates releasable according to the same rule. Figure 12 illustrates the client/server interaction that can be summarized as follows.

1. The client sends a request for a service to the server.
2. The server asks to the client a set of prerequisites, that is, a set of necessary conditions for granting access.
3. The client sends back the required prerequisites.
4. If the prerequisites are sufficient, than the server individuates the credentials and declarations needed to grant access to the resource.
5. The client evaluates the requests against its portfolio release rules and makes, eventually, some counter-requests.
6. The server sends back to the client the required certificates and declarations.
7. The client fulfills the server's requests.
8. The service is then granted to the client.

Since there may exist different policy combinations that may bring the access request to satisfaction, the communication of credentials and/or declarations could be an expensive task. To overcome this issue, the *abbreviation* predicates are used to abbreviate requests. Besides the necessity of abbreviations, it is also necessary for the server, before releasing rules to the client, to evaluate state predicates that involve private information. For instance, the client is not expected to be asked many times the same information during the same session and if the server has to evaluate if the client is considered or not trusted, it cannot communicate this request to the client itself.

Communication of requisites to be satisfied by the requester is then based on a filtering and renaming process applied on the server's policy, which exploits partial evaluation techniques in logic programs. The authors formally prove that the set of requirements that enable a service under the original policy coincides with the requirements specified by the filtering rules.

6 Policy Composition

In many real world situations, access control needs to combine restrictions independently stated that should be enforced as one, while retaining their independence and administrative autonomy. For instance, the global policy of a large organization can be the combination of the policies of its different departments and divisions as well as of externally imposed constraints (e.g., privacy regulations); each of these policies should be taken into account while remaining independent and autonomously managed. Policy composition is an orthogonal aspect with respect to the ones described in the previous sections, as policy composition should be independent from the languages adopted by each single entity.

In [9], the authors presented the following criteria that a composition framework for access control policies should satisfy.

- *Heterogeneous policy support.* The framework should support policies expressed in different languages and enforced by different mechanisms.
- *Support of unknown policies.* The framework should support policies that are not known a priori or that are only partially defined. Policies are therefore treated as black-boxes that can be queried at access control time and return a correct and complete response.
- *Controlled interference.* Policies cannot simply be merged as this may cause interferences and side effects. For instance, the accesses granted/denied might not correctly reflect the specifications anymore.
- *Expressiveness.* The language should support different methods for combining policies, without changing the input specifications and without ad-hoc extensions to authorizations.
- *Support of different abstraction levels.* The composition should highlight the different components and their interplay at different levels of abstraction.
- *Formal semantics.* The composition language should be declarative, implementation independent, and based on a solid framework to avoid ambiguity.

Various models have been proposed to reason about security policies [1,24,25, 35]. In [1,25] the authors focused on the secure behavior of program modules. McLean [35] proposed a formal approach including combination operators: he introduced *the algebra of security*, that is a Boolean algebra that enables to reason about the problem of policy conflict, arising when different policies are combined. However, even though this approach permits to detect conflicts between policies, it did not propose a method to resolve the conflicts and to construct a security policy from inconsistent sub-policies. Hosmer [24] introduced the notion of meta-policies (i.e., policies about policies), an informal framework for combining security policies. Subsequently, Bell [4] formalized the combination of two policies with a function, called *policy combiner*, and introduced the notion of *policy attenuation* to allow the composition of conflicting security policies. Other approaches are targeted to the development of a uniform framework to express possibly heterogeneous policies [7, 26, 27, 31, 48]. A different approach has been illustrated in [9] where the authors proposed an algebra for combining security policies together with its formal semantics. Following this work, Jajodia et al. [45] presented a propositional algebra for policies with a syntax consisting of abstract symbols for atomic policy expressions and composition operators. This framework has two classes of operators: *internal* and *external*. In the following, we will explain more in details the algebra for policy composition presented in [9].

6.1 An Algebra for Composing Access Control Policies

The need for a policy composition framework by which different component policies can be integrated while retaining their independence was first identified in [9]. Here, the authors propose an algebra of security policies together with its formal semantics and illustrate how complex policies can be formulated as expressions of the algebra. A policy is defined as a set of triples of the form (s,o,a), where s is a constant in (or a variable over) the set of subjects S, o is a constant in (or a variable over) the set of objects Obj, and a is a constant in (or a variable over) the set of actions A. Here, complex policies can then be obtained by combining policy identifiers, denoted P_i, through the *algebra operators*. The proposed algebra is parametric with respect to two languages: the *authorization constraint language*, used to specify the conditions under which a ground authorization is valid; and the *rule language*, used to state how a set of ground authorizations can be closed by deriving new authorizations from the ground set.

Algebra Syntax and Semantics

We are now ready to define the syntax and semantics of the algebra. Formally, the syntax is given by the following BNF grammar:

$$E ::= \mathbf{id} | E + E | E \& E | E - E | E \hat{} C | o(E, E, E) | E * R | T(E) | (E)$$
$$T ::= \tau \mathbf{id}.E$$

where **id** is a unique policy identifier, E is a policy expression, T is a construct, called *template*, C is a construct describing constraints, and R is a construct describing rules. The order of evaluation of operators is determined by the precedence which is (from higher to lower) τ, ., + and & and -, * and ^.

The semantics is a function mapping each policy expression in a set of ground authorizations and each template is a function over policies. Each policy identifier is mapped to sets of triples by *environments*.

Definition 3 (Environment). *An* environment e *is a partial mapping from policy identifiers to sets of authorization triples. By* $e[X/S]$ *we denote a modification of environment* e *such that*

$$e[X/S](Y) = \begin{cases} S & \textit{if } Y = X \\ e(Y) & \textit{otherwise} \end{cases}$$

The semantic of an identifier X in the environment e can be denoted as $[\![X]\!]_e = e(X)$. The operators of the algebra are defined as follows.

- *Addition* $(+)$. It merges two policies by returning their union.

  $$[\![P_1 + P_2]\!]_e = [\![P_1]\!]_e \cup [\![P_2]\!]_e$$

  Intuitively, additions can be applied in any situation where accesses can be authorized if allowed by any of the component policies (maximum privilege strategy).
- *Conjunction* $(\&)$. It merges two policies by returning their intersection.

  $$[\![P_1 \& P_2]\!]_e = [\![P_1]\!]_e \cap [\![P_2]\!]_e$$

  This operator enforces the minimum privilege strategy.
- *Subtraction* $(-)$. It deletes from a policy all the accesses in a second policy.

  $$[\![P_1 - P_2]\!]_e = [\![P_1]\!]_e \setminus [\![P_2]\!]_e$$

  Intuitively, subtraction specifies exceptions to statements made by a policy, and has the same functionalities of negative authorizations in existing approaches without introducing conflicts.
- *Closure* $(*)$. It closes a policy under a set of derivation rules.

  $$[\![P * R]\!]_e = \textbf{closure}(R, [\![P]\!]_e)$$

  The closure of policy P under derivation rules R produces a new policy that contains all the authorizations in P and those that can be derived evaluating R on P, according to a given semantics. The derivation rules in R can enforce, for example, an authorization propagation along a predefined subject or object hierarchy.
- *Scoping Restriction* $(\hat{\ })$. It restricts the applicability of a policy to a given subset of subjects, objects, and actions of the system.

  $$[\![P_1^\wedge c]\!]_e = \{t \in [\![P]\!]_e \mid t \text{ satisfy c}\}$$

  where c is a condition. It is useful to enforce authority confinement (e.g., authorizations specified in a given component can be referred only to specific subjects and objects).
- *Overriding* (o). It overrides a portion of policy P_1 with the specifications in policy P_2 and the fragment that is to be substituted is specified by a third policy P_3.

  $$[\![o(P_1, P_2, P_3)]\!]_e = [\![(P_1 - P_3) + (P_2 \& P_3)]\!]_e$$

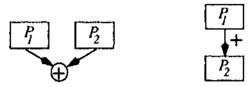
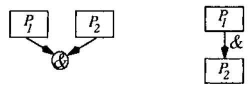
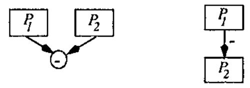
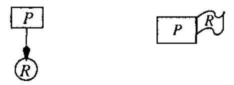
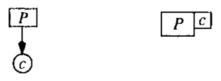
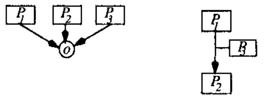
Operator	*Semantics* $[\![\ ]\!]_e$	*Graphical representation*
$P_1 + P_2$	$[\![P_1]\!]_e \cup [\![P_2]\!]_e$	
$P_1 \& P_2$	$[\![P_1]\!]_e \cap [\![P_2]\!]_e$	
$P_1 - P_2$	$[\![P_1]\!]_e \setminus [\![P_2]\!]_e$	
$P * R$	closure$(R, [\![P]\!]_e)$	
$P\hat{\ }c$	$\{t \in [\![P]\!]_e \mid t \text{ satisfy c}\}$	
$o(P_1,P_2,P_3)$	$[\![(P_1 - P_3) + (P_2 \& P_3)]\!]_e$	

Fig. 13. Operators of the algebra and their graphical representation

- *Template*(τ). It defines a partially specified (i.e., parametric) policy that can be completed by supplying the parameters.

$$[\![\tau X.P]\!]_e(S) = [\![P]\!]_{e[S/X]}$$

  where S is the set of all policies, and X is a parameter. Templates are useful for representing policies where some components are to be specified at a later stage. For instance, the components might be the result of further policy refinement, or might be specified by a different authority.

The algebraic operators just described have also a graphical representation summarized in Fig. 13.

The formal semantics on which the algebra is based allows us to reason about policy specifications and proves properties on them.

Evaluating Policy Expressions

Enforcement of compound policies is based on a translation from policy expressions into logic programs, which provide executable specifications compatible with different evaluation strategies. In particular, the following strategies can be applied:

E	$\mathsf{TR}(E,e)$
P	$\{\mathrm{auth}_P(s,o,a) \mid (s,o,a) \in e(P)\}$ if $e(P)$ is defined, $\emptyset$ otherwise.
$F +_i G$	$\{\mathrm{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z), \mathrm{auth}_i(x,y,z) \leftarrow mainp_G(x,y,z)\}$ $\cup \mathsf{TR}(F,e) \cup \mathsf{TR}(G,e)$.
$F \&_i G$	$\{\mathrm{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge mainp_G(x,y,z)\}$ $\cup \mathsf{TR}(F,e) \cup \mathsf{TR}(G,e)$.
$F -_i G$	$\{\mathrm{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge \neg mainp_G(x,y,z)\}$ $\cup \mathsf{TR}(F,e) \cup \mathsf{TR}(G,e)$.
$F\hat{}_i c$	$\{\mathrm{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge c\} \cup \mathsf{TR}(F,e)$.
$o_i(F,G,R)$	$\{\mathrm{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge \neg mainp_R(x,y,z),$ $\mathrm{auth}_i(x,y,z) \leftarrow mainp_G(x,y,z) \wedge mainp_R(x,y,z)\}$ $\cup \mathsf{TR}(F,e) \cup \mathsf{TR}(G,e) \cup \mathsf{TR}(R,e)$.
$F *_i R$	$\{\mathrm{auth}_i(s,o,a) \leftarrow \mathrm{auth}_i(s_1,o_1,a_1) \wedge .. \wedge \mathrm{auth}_i(s_n,o_n,a_n) \mid$ $((s,o,a) \leftarrow (s_1,o_1,a_1) \wedge \ldots \wedge (s_n,o_n,a_n)) \in R\}$ $\cup \{\mathrm{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z)\} \cup \mathsf{TR}(F,e)$.
$(\tau_i X.F)(G)$	$\{\mathrm{auth}_X(x,y,z) \leftarrow mainp_G(x,y,z)\} \cup \mathsf{TR}(F,e) \cup \mathsf{TR}(G,e)$.

Fig. 14. Translation TR: from policy expressions to logic programs

- *Materialization.* The policy expressions are evaluated thus determining the set of ground authorization terms corresponding to the accesses allowed by the policy. This strategy can be applied when all the individual policies are known and reasonably static.
- *Partial materialization.* Partial materialization can be considered mainly for two reasons. First, some of the component policies may be unknown at materialization time (black-box policies); clearly, such policies cannot be materialized. Second, some policies may be too dynamic to be materialized (as the cost of updating the materialization may exceed that of run-time evaluation).
- *Run-time evaluation.* This strategy enforces a run-time evaluation of each request (access triple) against the policy expression to determine whether the triple belongs to the result.

The authors propose a strategy, called *pe2lp*, for translating algebraic expressions into an equivalent logic program that is compatible with the different evaluation strategies above-mentioned. The logic program is then used for access control enforcement. Basically, the translation process creates a distinct predicate symbol for each policy identifier and for each algebraic operator in the expression. Since operators are not distinguishable, each of them is associated with a label, that is, an integer number associated from left to right and starting form 0. The result of this labeling process is a *canonical labeling* of the initial policy expression. Note that the *main label* of an expression is the integer associated with the outermost operator of the expression. Translation *pe2lp* takes a labeled policy expression and an environment as input and produces a logic program equivalent to the given expression. The translation process defines a predicate $\mathtt{auth}_P$, for each policy identifier P, and a predicate $\mathtt{auth}_i$, for each operator op_i. These predicates have three arguments: a subject, a resource, and an action. Figure 14 shows the translation of each operator. The *pe2lp* translation is semantic preserving, provided that the resulting program is interpreted according to the stable model semantics [22] or any other semantics equivalent to the stable model semantics on stratified programs. The logic programming formulation of algebra expressions can be used to enforce access control. First, for each *foreign policy* (i.e., policies expressed in different languages or stored at another site) a wrapper is needed that should be queried by the logic program [41]. The access control enforcement is then obtained by applying a materialization strategy, a partial materialization strategy, or a run-time strategy. In particular, partial materialization is obtained by applying standard partial evaluation techniques [40] to the logic program obtained by the translation process. It is important to highlight that partial evaluation preserves the meaning of the original logic program.

An interesting feature of the proposed algebra is that it can also be used to specify different elementary policies, such as the open or closed policies, or propagation rules along a hierarchy. To evaluate the expressiveness of the algebra, it can be useful a comparison with the *First Order Logic* (FOL). The composition algebra captures only a strict subset of the FOL because policy expressions refer to a well known fixed relation schema, corresponding to the authorization triple. In this way, the *containment* decision problem (P_1 is contained in P_2) and the *checking strong equivalence* (P_1 and P_2 are exactly equivalent) are decidable for policy expressions. As a result of the comparison between FOL and the algebra we have that:

- closure-free policy expressions capture exactly the quantifier-free 0-1 fragment[6] of monadic first-order logic;
- quantifiers can be captured with the closure operator and one simple rule.

The first-order language is induced by predicates $\{P_{all}, P_1, P_2 \ldots\}$, representing policy identifiers (P_{all} denotes the set of all authorization triples), and $\{C_1, C_2 \ldots\}$, representing constraints.

It is important to note that, from the basic domains S, Obj, and A, from the interpretation of constraint predicates, $\mathsf{satisfy}$, and from an environment e, the interpretation structures for the monadic first-order logic just introduced are of the form: ($\mathsf{S} \times \mathsf{Obj} \times \mathsf{A}$, e, **satisfy**), denoting that triple (s, o, a) is or not an authorization for environment e.

As an example of policy composition, consider the scenario introduced in Sect. 3.1 and suppose that the computer on-line store is composed of three depart-

[6] A *0-1 formula* F is a formula where each sub-formula of F has at most one free variable.

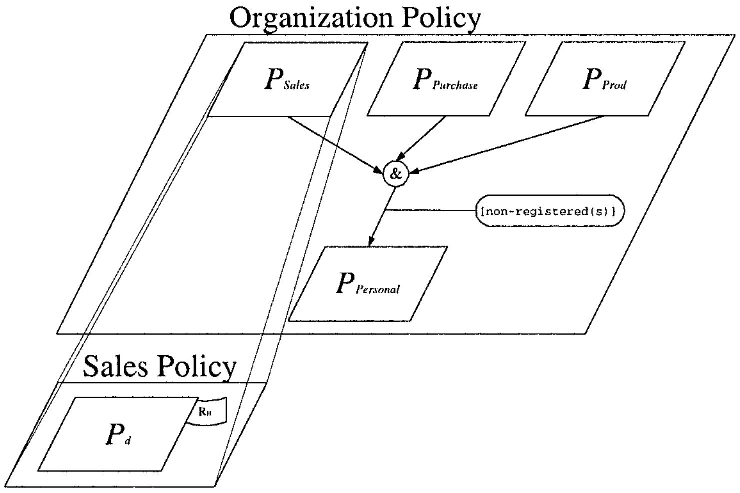


Fig. 15. An example of policy composition

ments, named *Purchase*, *Sales*, and *Production*. The manager of each department is responsible for granting access to data under his responsibility. Let $P_{Purchase}$, P_{Sales} and P_{Prod} be the policies of the three departments. Suppose now that an access is authorized if any of the department policies state so and that authorizations in policy P_{Sales} are propagated to individual users and documents by classical hierarchy-based derivation rules, denoted R_H. Also, suppose that to access the on-line store, non-registered users need also the *Personal* manager consent, stated by policy $P_{Personal}$. In terms of the algebra, the computer store policy can be represented as:

$$o(P_{Purchase}\&P_{Sales} * R_H\&P_{Prod}, P_{Personal}, (P_{Purchase}\&P_{Sales} * R_H\&P_{Prod})^{\wedge}(non - registered(s)))$$

Figure 15 reports the graphical representation of the computer on-line store policy.

While this algebra is expressive and powerful, it leaves space for further work. Future work to be carried out includes investigation of administration policies for regulating the specification of the different component policies by different authorities; the analysis of incremental approaches to enforce changes to component policies; the analysis of mobile policies, that is, policies associated with objects and that follow the objects when they are passed to another site. Because different and possibly independent authorities can define different parts of the mobile policy in different time instants, the policy can be expressed as a policy expression. In such a context, there is the problem on how to ensure the obedience of policies when the associated objects move around.

7 Conclusions

An important requirement of any system is to protect its data and resources against unauthorized disclosure and/or improper modifications, while at the same time ensuring their availability to legitimate users. A fundamental component in enforcing protection is represented by the access control service whose task is to control every access to a system and its resources and ensure that all and only authorized accesses can take place. Throughout the chapter we presented the basic concepts of access control and investigated different issues concerning the development of an access control system, discussing recent proposals in the area of access control models and languages.

8 Acknowledgments

This work was supported in part by the European Union within the PRIME Project in the FP6/IST Programme under contract IST-2002-507591 and by the Italian MIUR within the KIWI and MAPS projects.

References

1. Abadi M, Lamport L (1992). Composing specifications. ACM Transactions on Programming Languages, 14(4):1–60.
2. Ardagna CA, Damiani E, De Capitani di Vimercati S, Samarati P (2004). XML-based access control languages. Information Security Technical Report.
3. Atkinson B, Della Libera GD, et al. (2002). Web services security (WS-Security). http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-security.asp.
4. Bell D (1994). Modeling the multipolicy machine. In Proc. of the New Security Paradigm Workshop, Little Compton, Rhode Island, USA.
5. Bertino E, Bettini C, Ferrari E, Samarati P (1998). An access control model supporting periodicity constraints and temporal reasoning. ACM Transactions on Database Systems, 23(3):231–285.
6. Bertino E, Bonatti P, Ferrari E (2001). TRBAC: a temporal role-based access control method. ACM Transactions on Information and System Security, 4(3):191–223.
7. Bertino E, Jajodia S, Samarati P (1999). A flexible authorization mechanism for relational data management systems. ACM Transactions on Information Systems, 17(2):101–140.
8. Blaze M, Feigenbaum J, Lacy J (1996). Decentralized trust management. In Proc. of the 1996 IEEE Symposiumon Security and Privacy, Oakland, CA, USA.
9. Bonatti P, De Capitani di Vimercati S, Samarati P (2002). An algebra for composing access control policies. ACM Transactions on Information and System Security, 5(1):1–35.
10. Bonatti P, Samarati P (2002). A unified framework for regulating access and information release on the web. Journal of Computer Security, 10(3):241–272.
11. Box D, et al. (2003). Web services policy assertions language (WS-PolicyAssertions) version 1.1. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-policyassertions.asp.

12. Box D, et al. (2003). Web Services Policy Attachment (WS-PolicyAttachment) version 1.1. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-policyattachment.asp.
13. Box D, et al. (2003). Web services policy framework (WS-Policy) version 1.1. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-policy.asp.
14. Damiani E, De Capitani di Vimercati S, Paraboschi S, Samarati P (2000). Securing XML documents. In Proc. of the 2000 International Conference on Extending Database Technology (EDBT2000), Konstanz, Germany.
15. Damiani E, De Capitani di Vimercati S, Paraboschi S, Samarati P (2002). A fine-grained access control system for XML documents. ACM Transactions on Information and System Security, 5(2):169–202.
16. DeTreville J (2002). Binder, a logic-based security language. In Proc. of the 2001 IEEE Symposium on Security and Privacy, Oakland, CA, USA.
17. eXtensible Access Control Markup Language (XACML) Version 2.0 (2004). eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS. http://www.oasis-open.org/committees/xacml.
18. Farrell S, Housley R (2002). An internet attribute certificate profile for authorization. RFC 3281.
19. Ferraiolo D, Kuhn R (1992). Role-based access controls. In Proc. of the 15th NIST-NSA National Computer Security Conference, Baltimore, Maryland.
20. Gabillon A (2004). An authorization model for XML databases. In Proc. of the ACM Workshop Secure Web Services, George Mason University, Fairfax, VA, USA.
21. Gabillon A, Bruno E (2001). Regulating access to XML documents. In Proc. of the Fifteenth Annual IFIP WG 11.3 Working Conference on Database Security, Niagara on the Lake, Ontario, Canada.
22. Gelfond M, Lifschitz V (1988). The stable model semantics for logic programming. In Proc. of the 5th International Conference and Symposium on Logic Programming, Cambridge, Massachusetts.
23. Gladman B, Ellison C, Bohm N (1999). Digital signatures, certificates and electronic commerce. http://jya.com/bg/digsig.pdf.
24. Hosmer H (1992). Metapolicies II. In Proc. of the 15th National Computer Security Conference, Baltimore, MD.
25. Jaeger T (2001). Access control in configurable systems. Lecture Notes in Computer Science, 1603:289–316.
26. Jajodia S, Samarati P, Sapino ML, Subrahmanian VS (2001). Flexible support for multiple access control policies. ACM Transactions on Database Systems, 26(2):214–260.
27. Jajodia S, Samarati P, Subrahmanian VS, Bertino E (1997). A unified framework for enforcing multiple access control policies. In Proc. of the 1997 ACM International SIGMOD Conference on Management of Data, Tucson, AZ.
28. Jim T (2001). Sd3: A trust management system with certified evaluation. In Proc. of the 2001 IEEE Symposium on Security and Privacy, Oakland, CA, USA.
29. Kudoh M, Hirayama Y, Hada S, Vollschwitz A (2000). Access control specification based on policy evaluation and enforcement model and specification language. In Symposium on Cryptograpy and Information Security (SCIS'2000), Japan.
30. Landwehr CF (1981). Formal models for computer security. ACM Computing Surveys, 13(3):247–278.
31. Li N, Feigenbaum J, Grosof B (1999). A logic-based knowledge representation for authorization with delegation. In Proc. of the 12th IEEE Computer Security Foundations Workshop, Washington, DC, USA.

32. Li N, Grosof B, Feigenbaum J (2003). Delegation logic: A logic-based approach to distributed authorization. ACM Transactions on Information and System Security, 6(1):128–171.
33. Li N, Mitchell JC (2003). Datalog with constraints: A foundation for trust-management languages. In Proc. of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003), New Orleans, LA, USA.
34. Li N, Mitchell JC, Winsborough WH (2002). Design of a role-based trust-management framework. In Proc. of the IEEE Symposium on Security and Privacy, Oakland, CA, USA.
35. McLean J (1988). The algebra of security. In Proc. of the 1988 IEEE Computer Society Symposium on Security and Privacy, Oakland, CA, USA.
36. Ryutov T, Zhou L, Neuman C, Leithead T, Seamons KE (2005). Adaptive trust negotiation and access control. In Proc. of the 10th ACM Symposium on Access Control Models and Technologies, Stockholm, Sweden.
37. Samarati P, De Capitani di Vimercati S (2001). Access control: Policies, models, and mechanisms. In Focardi R, Gorrieri R, editors, Foundations of Security Analysis and Design, LNCS 2171. Springer-Verlag.
38. Seamons KE, Winsborough W, Winslett M (1997). Internet credential acceptance policies. In Proc. of the Workshop on Logic Programming for Internet Applications, Leuven, Belgium.
39. Security Assertion Markup Language (SAML) V1.1 (2003). Security Assertion Markup Language (SAML) V1.1. OASIS. http://www.oasis-open.org/committees/security/.
40. Sterling L, Shapiro E (1997). The art of Prolog. MIT Press, Cambridge, MA.
41. Subrahmanian V, Adali S, Brink A, Lu J, Rajput A, Rogers T, Ross R, Ward C. Hermes: heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/hermes.
42. The XACML Profile for Hierarchical Resources (2004). The XACML Profile for Hierarchical Resources. OASIS. http://www.oasis-3893open.org/committees/xacml.
43. van der Horst TW, Sundelin T, Seamons KE, Knutson CD (2004). Mobile trust negotiation: Authentication and authorization in dynamic mobile networks. In Proc. of the Eighth IFIP Conference on Communications and Multimedia Security, Lake Windermere, England.
44. Web services security policy (WS-SecurityPolicy) (2002). Web services security policy (WS-SecurityPolicy). http://www-106.ibm.com/developerworks/library/ws-secpol/.
45. Wijesekera D, Jajodia S (2003). A propositional policy algebra for access control. ACM Transactions on Information and System Security, 6(2):286–325.
46. Winsborough W, Seamons KE, Jones V (2000). Automated trust negotiation. In Proc. of the DARPA Information Survivability Conf. & Exposition, Hilton Head Island, SC, USA.
47. Winslett M, Ching N, Jones V, Slepchin I (1997). Assuring security and privacy for digital library transactions on the web: Client and server security policies. In Proc. of the ADL '97 — Forum on Research and Tech. Advances in Digital Libraries, Washington, DC.
48. Woo TYC, Lam SS (1993). Authorizations in distributed systems: A new approach. Journal of Computer Security, 2(2,3):107–136.
49. World Wide Web Consortium (W3C) (2004). eXtensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium (W3C). http://www.w3.org/TR/REC-xml.
50. Yu T, Ma X, Winslett M (2000). An efficient complete strategy for automated trust negotiation over the Internet. In Proc. of the 7th ACM Computer and Communication Security, Athens, Greece.

51. Yu T, Winslett M (2003). A unified scheme for resource protection in automated trust negotiation. In Proc. of the IEEE Symposium on Security and Privacy, Berkeley, California.
52. Yu T, Winslett M, Seamons KE (2001). Interoperable strategies in automated trust negotiation. In Proc. of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania.
53. Yu T, Winslett M, Seamons KE (2003). Supporting structured credentials and sensitive policies trough interoperable strategies for automated trust. ACM Transactions on Information and System Security, 6(1):1–42.