October 26, 1999
RT0331
Security 34 pages

Research Report

A Comparative Survey of Authorization Languages

Anish Mathuria

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Comparative Survey of Authorization Languages

Anish Mathuria

IBM Tokyo Research Laboratory, LAB-S77
1623-14, Shimotsuruma, Yamato-shi
Kanagawa-ken 242-0001, JAPAN
E-mail: anish@trl.ibm.co.jp

October 26, 1999

(Research Report)

A Comparative Survey of Authorization Languages

Abstract

This paper gives a comparative survey of selected authorization languages proposed in the literature. It attempts to classify authorization languages by analogy with domain-specific languages. The languages are classified into one of the following domains: (1) operating systems authorization, (2) database authorization, (3) temporal authorization, (4) hypertext authorization, (5) certificate authorization, (6) role-based authorization, and (7) computer supported cooperative work authorization. For each domain, some prominent example languages are described in an informal yet uniform way. Throughout the paper, comparisons are made to highlight the similarities and differences between various languages.

1 Introduction

Authorization policies essentially describe how access to information and resources is regulated. They are generally designed in order to meet high level security policies which individuals or organizations wish to have as protection mechanisms. Most of the early work on authorization was in the context of operating systems. Since then, a great many authorization languages have been created which are tailored to other domains. The aim of this survey is to classify and compare various authorization languages which have been proposed in recent years. It approaches authorization languages by analogy with domain specific languages—that is, languages which allow programs for particular application domains to be written using more specialized constructs than that afforded by general purpose languages. A textbook discussion on domain-specific languages is offered by Hudak [1, Chapter 3].

The following major domains are identifiable in respect of authorization languages:

- Operating systems authorization;
- Database authorization;
- Temporal authorization;
- Hypertext authorization;
- Certificate authorization;
- Role-based authorization;
- CSCW authorization.

Although the above list is perhaps an oversimplification, it seems sufficient to cover a lot of existing languages. However, it may be necessary to add other domains in order to cover more ground. It is also worth noting that there exist authorization languages which span more than one domain. For example, although temporal authorization can be studied in its own right, temporal authorization languages as such have appeared in the

context of database authorization (as in Bertino, Bettini, Ferrari and Samarati [14]) and workflow authorization (as in Atluri and Huang [15]), amongst others.

Despite the wide variety of authorization languages, the common design goal behind all such languages is to specify and define access control policies. Usually, the entities that are authorized by a policy are called *subjects*. The resources that are the target of access by subjects are called *objects*. To ease the task of describing and comparing different languages, we need to look at access control policies from an abstract point of view. Following the framework of Abrams, Eggers, La Padula and Olson [2], access control policies can be viewed in terms of the following components:

Access Control Information (ACI): This component specifies information associated with subjects and objects.

Access Control Context (ACC) This component specifies additional information not generally associated with subjects and objects.

Access Control Rules (ACR) This component specifies the conditions under which access to objects is granted to subjects.

Access Control Authorities (ACA): This component specifies who has the right to change the information specified in ACI, ACC, or ACR.

Note that the last component regulates access to policies themselves, thus it provides meta access control. In what follows, we will use the framework of Eggers *et al* to describe some prominent example authorization languages that have been created for various domains.

2 Database authorization languages

A considerable number of languages have been proposed for authorization in relational or object oriented databases. A characteristic features of database authorization is the use of *implicit authorizations*. The basic idea behind it is that an explicitly specified authorization may imply other authorizations (implicitly). While this idea was first used in the context of database authorization, it has found wide use in other contexts. Implicit authorizations are naturally expressed via implication rules, thereby reducing the number of authorizations that need to be explicitly defined. Implicit authorizations can generally be distinguished as *data-model oriented* or *group oriented*. For example, in the relational model, an update privilege on a record should imply a read privilege on the record, regardless of the authorization subject or object. The authorization here can be classified as data-model oriented. Implicit authorization that are derived on the basis of subject, object, or privilege aggregation can be classified as group-oriented. For example, a read authorization for a group of users on a certain database object intuitively implies a read authorization on the object for any member of the group of users.

We can distinguish between two kinds of database authorization languages. There exist languages which assume a fixed data model, for example the relational or object-oriented data model. The basic principle behind these languages is that the authorizations inherent to the data model can be handled automatically without requiring the user to explicitly specify them. This allows to minimize the explicit attention which a policy writer must pay to program the desired access control policy. An alternative principle is to make

access control intentions explicit by requiring the data model dependent authorizations to be specified by the policy writer. There exist database authorization languages whose design follows this principle.

Most of the database authorization languages proposed in recent years allow both positive and negative authorizations to be specified. In essence, a positive authorization specifies the granting of access right, whereas a negative authorization specifies the denial of access right. The ability to specify negative authorization provides greater flexibility to the policy writer in specifying policies. Following Jajodia, Samarati and Subrahmanian [4], we can divide the approach taken to regulate the coexistence of negative and positive authorizations into the following two components:

Derivation component This component regulates how authorizations of subject groups propagate to their members. Any inconsistencies which arise, when a subject belongs, either directly or indirectly, to groups with conflicting authorizations may be handled by this component, depending on the derivation policy.

Conflict resolution component This component acts as the ultimate source of authority for solving conflicts.

As we will see below, different languages generally make different choices with respect to policies for conflict resolution and derivation.

2.1 Relational database authorization

An example of a recently proposed authorization language for relational databases is that of Bertino, Jajodia and Samarati [6] (BJS).

ACI

Subjects of authorizations can be either users or groups. Groups can be nested, provided cycles are not introduced. An object corresponds to a database table.

ACR

An access control rule is specified with respect to a subject, privilege, authorization sign, table, authorization issuer, and authorization type. The authorization sign "+" ("-") indicates a positive (negative) authorization. The authorization type *weak* indicates an authorization that does not allow for exceptions. The authorization type *strong* means the opposite.

ACA

The administrative policy is based on ownership together with decentralized administration of authorizations. The creator of a table is defined to be the table owner. He can grant others privileges to the table. The model of administration divides privileges into two classes: (1) An *access privilege* refers to the right to some operation on the table. (2) An *administrative privilege* refers to the right to grant authorizations on the table. (The idea that the right to grant authorizations can itself be treated as a right is originally found in the database authorization model proposed by Griffiths and Wade [3] in 1976.)

We can thus distinguish authorizations as either access authorizations or administrative authorizations. Two kinds of administrative privileges are considered. (1) The *àdm-access* privilege allows subjects to grant/revoke access authorizations on the table. (2) The *àdminister* privilege allows subjects to grant/revoke access as well as administrative authorizations on the table. The model does not separate the administration of positive and negative authorizations. Unlike access authorizations, administrative authorizations are not classified as strong or weak. Allowing weak administrative authorizations implies that when an administrative authorization granted to a user becomes overridden the overriding effect has to be propagated to all authorizations granted by the user. This would make the model unnecessarily complicated. An *administrative authorization* is specified with respect to a subject, access privilege, administrative privilege, authorization type, table and authorization issuer. Here, the authorization type does not refer to the administrative authorization itself. Rather, it refers to the authorization type of authorizations that can be issued by the subject.

As discussed below, the language makes a fixed choice is made with respect to the derivation and conflict resolution components.

2.1.1 Derivation component

If a subject belongs, either directly or indirectly, to groups with conflicting authorizations, then depending on how these groups are connected in the subject hierarchy, one of the following overriding rule is used to determine which authorization wins:

- *Strong-auth overrides:* A strong authorization takes precedence over a weak authorization.
- *Sub-subject overrides:* An authorization specified for a subject takes precedence over an authorization specified for a supersubject.
- *Path overrides:* An authorization specified for a subject s takes precedence over an authorization specified for a supersubject of s only along the paths passing from s.

To specify the derivation component, fix three subjects, say s_0, s_1 and s_2, such that s_0 is a member of both s_1 and s_2. Let a_1 and a_2 be authorizations explicitly granted to s_1 and s_2, respectively, and assume that a_1 and a_2 carry contrasting signs.

The derivation component is summarized in Table 1, where the symbol $\ni$ is used to indicate group membership.

Conflicting Auth.		*Subject relationships*	*Overriding rule*	*Overriding*
a_1	a_2		*wrt* s_0	*Auth.*
strong	weak	$s_0 \in s_1$, $s_0 \in s_2$	strong-auth overrides	a_1
weak	weak	$s_1 \in s_2$, $s_0 = s_1$	sub-subject overrides	a_1
weak	weak	$s_1 \in s_2$, $s_0 \in s_1$, $s_0 \in s_2$	path-overrides	a_1

Table 1: Derivation component

2.1.2 Conflict resolution component

Note that the overriding rules given in Table 1 do not solve conflicts between two weak authorizations for which s_1 and s_2 are not related by the membership relation. These are resolved by the conflict resolution component. If a conflict between two weak authorizations cannot be resolved using the above rules, then both authorizations are considered invalid. Finally, the presence of two conflicting strong authorizations is considered an inconsistency and, therefore, not accepted by the system. An algorithm for detecting conflicts between strong authorization is built into the system.

The conflict resolution component is summarized in Table 2.

Conflicting Auth.		*Solution*
weak	weak	nil
strong	strong	not appl

Table 2: Conflict resolution component

2.2 Objected oriented database authorization

One of the earliest and most influential work in this area is that of Rabitti, Bertino, Kim and Woelk [7]. As emphasized by them, the use of implicit authorizations is especially applicable to the rich data models supported in object-oriented databases. For example, it is useful to defined a single authorization on a class object and have the same authorization apply implicitly to all class instance objects. A brief review of basic object-oriented concepts is in order.

An *object* has two basic properties: state and behavior. The state represents the values for the attributes of the object. The behavior represents the set of methods that operate on the state. A *class* is an object which essentially characterizes all objects that share the same set of attributes and methods. Classes can be nested, that is, the attributes of a class may themselves be classes. The nesting results in a directed graph of classes, called class-composition hierarchy.

An existing class C can be used to define a new class C', whose set of attributes and methods is a superset of the corresponding set for the class C. The class C' is then said to inherit attributes and methods from the class C. The class C is called a superclass of C'; conversely, C' is called a subclass of C. Single inheritance means that a subclass is allowed to have at most one superclass. Multiple inheritance means that a subclass is allowed to have more than one superclass. The inheritance relationship between classes results in a hierarchy, called class hierarchy.

ACI

Subjects can be *roles*. In essence, a role can be thought of as a named job function within an organization [8]. Roles can be nested, resulting in a *role hierarchy*. If there is an arc from a role r to a role r' in the role hierarchy, then r can be thought of as a role that is superior to role r'. Each object has a unique identifier. The set of operations that can be performed on objects includes: R (Read), W (Write), G (Generate) and RD (Read

definition). The operation G allows to create an object. The operation RD allows to read the definition of an object.

ACR

An authorization is specified with respect to a subject, object and authorization type. Here, any authorization type should be understood as a privilege. If the symbol $\neg$ is prefixed to an authorization type, it indicates a negative authorization; otherwise, the authorization is positive. Authorizations are defined as weak or strong, depending on whether they can be overridden or not. Different types of braces are used to syntactically differentiate between strong and weak authorizations. If s is a subject, o is an object, and a is an authorization type, then (s, o, a) indicates a strong authorization, whereas $[s, o, a]$ indicates a weak authorization.

Implicit authorizations are deduced from explicitly defined authorizations along each of the three authorization dimensions—that is, subject, object and authorization type. We consider each category in turn below.

Subject implication rules: Implicit authorizations along the subject dimension are derived by the following rule. If there is an arc from role r to role r' in the role hierarchy, then:

1. the role r implicitly receives all positive authorizations of role r';
2. the role r' implicitly receives all negative authorizations of role r.

The above rules ensures that the authorization for role r subsume the authorizations for role r'.

Authorization type implication rules: An implication relationship, $>$, is used to derive implicit authorizations along the authorization type dimension. The definition of $>$ includes the following: $W > R$, $W > G$, $G > RD$ and $R > RD$. For example, $G > RD$ captures the intuition that a user who can create object instances must also be able to read the class definition.

Object implication rules: The deduction of implicit authorizations along the object dimension is based on the concepts of *Authorization Object Schema* (AOS) and *Authorization Object Lattice* (AOL). The AOS represents the different object types in a database. An example of an object type is the class definition type. The idea behind the AOS is that an authorization for a particular object type may imply the same authorization for another object type. The AOL represents the various authorization objects in a database. Each node in AOL is restricted to exactly one authorization object type in AOS. Directed arcs in AOL define implication links between nodes, reflecting implicit authorizations between the node object types. For example, the object types *Class* and *Instance* are connected via an implication link in the AOS. This link indicates that an authorization on a class implies an authorization on all objects that are instances of that class. The authorization types that propagate down the AOL are defined by the set $A.down$; this set includes the types R and W. The types that propagate up the AOL are defined by the set $A.up$; this set includes the type RD. The authorization types that do not propagate

up or down the AOL are defined by the set *A.nil*; this set includes the type G. Implicit authorizations on objects along the AOL are derived according to the following rule. If there is an implication link from object o to object o' in the AOL, then:

1. the object o' implicitly receives all positive (respectively, negative) authorizations of o, whose authorization types are in *A.down* (respectively, *A.up*);

2. the object o receives all positive (respectively, negative) authorizations of o, whose authorization types are in *A.up* (respectively, *A.down*).

Authorizations on the attributes of a class are modelled by introducing authorization objects *Setof-Attributes* and *Attribute* as part of the AOS. Similarly, authorizations on the methods of a class are modelled by introducing authorization objects *Setof-Methods* and *Method*. Both kinds of authorizations are restricted to object instances; they do not apply to the definition of attributes and methods. One issue is whether the creator of a class has any implicit authorizations on the instances of a subclass derived from the class. The design choice of Rabitti *et al* is that he does not implicitly receive any authorizations of this kind. An authorization type, SG (subclass generate), is defined to control the creation of subclasses. The semantics of this type is defined by the following relationships: $W > SG$ and $SG > RD$. The authorization object schema is further extended to allow authorizations with regards to two additional object-oriented database concepts: composite objects and versioned objects.

2.3 ASL

The Authorization Specification Language, or ASL, of Jajodia, Samarati and Subrahmanian [4] assumes a generic data model. Their language offers greater flexibility in the choice of the data model at the price that implicit authorizations are not handled automatically.

ACI

Subjects of authorizations can be either users, groups, or roles. Groups can be nested, provided cycles are not introduced. Objects of authorizations depend on the data model.

ACC

A history of execution of accesses is maintained as a collection of Prolog-style facts, called *done* facts. The fact *done(o, u, R, a, t)* holds if a user u with possibly more than one role active, collectively defined as the set R, has executed action a on object o at time t.

ACR

The access control rules are specified in terms of Prolog-like predicates. Two fixed classes of predicates are provided. The first class consists of predicates for which a policy writer is allowed to define rules, subject to certain restrictions. The second class consists of built-in predicates with pre-defined meaning. The policy writer cannot define rules for these predicates. He can only call these predicates when defining rules for the former class of predicates. We describe each class of predicates in turn below.

The predicates *cando/3* and *dercando/3* have the purpose of specifying base and derived authorizations, respectively. The predicate *do/3* has two purposes. The first is to resolve conflicts when the rules for *cando* and *dercando* result in both positive and negative authorizations for a given object, subject, and action. The second is to force a default decision in the situation that the rules for *cando* and *dercando* do not imply a positive or negative authorization for a given object, subject, and action. The predicate *grant/4* has the purpose of specifying whether a user with possibly more than one role active is to be allowed or denied access.

The built-in predicates include *done/5*, *active/2*, *dirin/2*, *in/2* and *typeof/2*. Their meanings are summarized in Table 3.

Predicate	**Meaning**
$done(o, u, R, a, t)$	User u with roles in R active has executed action a on object o at time t
$active(u, r)$	User u has role r active
$dirin(s_1, s_2)$	s_1 is a direct member of s_2
$in(s_1, s_2)$	s_1 is an indirect member of s_2
$typeof(o, t)$	o is of type t

Table 3: Built-in predicates in ASL

Any built-in predicate, i.e. `done`, `active`, `dirin`, `in`, or `typeof`, can be called when specifying rules for a non built-in predicate. The non built-in predicates that can be called are summarized in Table 4, where the horizontal rows represent caller predicates and the columns represent callee predicates.

	done	*cando*	*dercando*	*do*	*grant*	*error*
cando	No	No	No	No	No	No
dercando	Yes	Yes	Yes	No	No	No
do	Yes	Yes	Yes	No	No	No
grant	Yes	Yes	Yes	Yes	No	No
error	Yes	Yes	Yes	Yes	Yes	No

Table 4: Restrictions on rules for non built-in predicates

The above restrictions are designed to ensure that every policy that can be expressed using the language is a special type of logic program, called *stratified datalog program* (for a definition, see Ullman [5, Chapter 3]). This reduction allows some well-known results from logic programming theory to be adopted. Two important consequences are that ASL has a well-defined semantics and that it can be evaluated efficiently.

A specification correctness criteria is formulated in terms of consistency and completeness requirements. An authorization specification is *consistent* if it does not simultaneously grant and deny any access request; it is *complete* if it always yields an access decision, i.e., either grant or deny, for all possible access requests. The correctness re-

quirement can be enforced by specifying fixed rules for the *error* predicate. Checking the correctness of a specification therefore entails policy evaluation.

The following high level conflict resolution and derivation policies can be expressed using explicit rules in ASL. The choice of what policies are used is left up to the policy writer.

Conflict resolution: no conflict, denials take precedence, and permissions take precedence, nothing takes precedence.

Derivation: no overriding, sub-subject overriding, and path overrides.

2.4 FAM/CAM languages

The FAM/CAM languages proposed by Jajodia, Samarati, Subrahmanian and Bertino [9] are closely related to the ASL language. The basic constructs of FAM/CAM are similar to that of ASL. However, there are some notable differences:

- Roles are not included in FAM/CAM. As a general rule, the FAM/CAM counterpart to an ASL predicate is obtained by omitting the role-term argument if present.
- Access decisions are specified by a new predicate, *do/3*, rather than by the *grant* predicate as in ASL. The *grant* predicate is not included in FAM/CAM.
- An additional built-in predicate is provided: $owner(o, s)$ means s is owner of o.
- The built-in predicate *done* is simplified by deleting the argument for time.
- The definition of the *error* predicate is modified to allow arguments for object, subject and action.

The distinguishing feature of the FAM/CAM design is that it divides policy inconsistency into generic or specific. A generic inconsistency is avoided by imposing restrictions at the specification level. An application specific inconsistency is derived using error rules. Note that in ASL both types of inconsistencies are prevented using *error* rules.

FAM programs are less tightly specified as compared to CAM programs. The former permit both positive and negative authorizations, whereas the latter only permit positive authorizations. As a matter of fact, it is possible to define a FAM program which neither grants nor denies an access request. FAM programs are not guaranteed to be complete and, therefore, only partially correct with respect to the specification correctness criteria used in ASL. A CAM program, on the other hand, is guaranteed to be complete, by definition. All CAM programs implement the closed world model. That is, if no positive authorization is implied, then access is denied.

2.5 Comparison

As a general rule, database access control can be viewed as a *boolean* function which, given an access request and a policy, returns *true* if the request is authorized by the policy and *false* if it is denied by the policy. The above view assumes that policies satisfy the consistency and completeness requirements. Two typical approaches have been used by designers of database authorization languages for ensuring consistency and completeness.

These can be distinguished as enforcement by *specification level check* or *access control level check*. A specification level check is essentially syntactic in nature; it does not require policy evaluation. In contrast, an access control level check requires policy evaluation.

Although ASL and FAM support exceptions by the way of negative authorizations, they do not allow exceptions to be stated at the level of individual authorizations. Hence these languages only provided limited support for exceptions, as compared to BJS.

Tables 5, 6 and 7 give a comparison of BJS/ASL/FAM/CAM.

Language	*Pos. and Neg. Auth.*	*Exceptions/ Strong Enforc.*	*Admin.*	*Subject grouping*
BJS	Yes	Yes/Yes	Yes	groups
ASL	Yes	limited/no	No	roles
FAM	Yes	limited/no	No	groups
CAM	positive only	not appl/not appl	No	groups

Table 5: General features

Language	*Consistency enforcement*	*Completeness enforcement*
BJS	spec. check	acc. check
ASL	acc. check	acc. check
FAM	spec. check	not appl
CAM	spec. check	acc. check

Table 6: Correctness enforcement

Language	*Conflict resolution component*	*Derivation component*
BJS	built-in	built-in
FAM	user specified	user specified
CAM	not appl	not appl
ASL	user specified	user specified

Table 7: Conflict resolution and derivation components

3 Operating systems authorization languages

The *need-to-know policy* (sometimes called the *principle of least privilege* [12]) is central to the design of early languages for operating systems authorization. In effect, it requires

that all users and programs operate with the smallest set of privileges necessary to perform their functions, a sound principle for any system. A basic design rule for supporting the need-to-know policy is to have protection mechanisms based on permissions rather than exclusion. Many languages for operating systems authorization make use of this design principle and require the policy writer to express the protection requirements in terms of what should be allowed, implicitly denying all rights that are not explicitly allowed. This is often called the *closed world* model. The *open world model*, where all rights are implicitly allowed unless explicitly denied, is inherently less secure than a closed world system.

The operating systems authorization languages are distinguished from other kinds of languages by their use of *access control lists*. An access control list (acl) is basically associated with each object to be protected. Typically, an acl consists of a list of positive authorizations; an entry (s, R) in an acl for object o means that subject s is granted set of access rights R to o. In languages that do support negative authorizations, the intended role of such authorizations is generally limited to specifying exceptions. The support for negative authorizations varies from none (the weakest), through denial at the level of all access rights, up to denial at the level of individual access right (the strongest).

The use of access control lists together with limited support for negative authorizations permit efficient evaluation algorithms to be constructed. One evaluation rule that is often used has the property that the first access list entry that matches a request overrides all other entries. As a consequence, not all entries in the access list may need to be examined. This rule speeds up authorization evaluation but forces conflict resolution based on ordering.

3.1 GACL language

The Generalized Access Control List language, or GACL, of Woo and Lam [10] extends conventional ACL-based languages by providing rules to express conditional restrictions on access rights.

ACI

Subjects can be users, groups, or in general some expression. A minus sign can be prefixed to a user or group name to indicate a complementary subject. A compound subject is formed using the conjunction operator, '$\wedge$', and is constructed by delegation. It represents a subject who has authority to act as each of its component subjects. In GACL, objects are modelled by a set of object identifiers and operations by a set of operation identifiers. The actual definitions of these sets is up to the policy writer.

ACC

In GACL, predicate symbols are used to model system conditions that are relevant to access control decisions.

ACR

Three kinds of access control rules can be specified. The rules support both positive and negative authorizations. If a minus sign is used in front of an operation name, then it

indicates a denial. For example, -read means not allowed to read.

- A *closure rule*, expressed via the $\rightarrow$ construct, is essentially an if-then-style rule. It allows to specify authorizations which are implied by other authorizations.

- An *inheritance rule*, expressed via the **inherit** keyword, is basically a variation on the closure rule. Two kinds of inheritance rules are defined, depending on the keyword specified in front of **inherit**. If the keyword **always** is placed in front, the inheritance rule essentially becomes a shorthand for the following form of closure rule,

$$obj ::< [\_x], [\_r] > \Rightarrow < [\_x], [\_r] >$$

  Here $\_x$ and $\_r$ denote variables of the appropriate type and *obj* names some object other than the current object, that is, the object whose acl contains the inheritance rule. This form of inheritance rule means that any subject who is allowed (denied) certain operations on object *obj* inherits the same permissions (denials) on the current object. If, on the other hand, the keyword **demand** is placed in front of **inherit**, the rule applies only if no previous entries have authorizations which grant or deny the inherited rights.

- A *default rule*, expressed via the **default** keyword, is similar in spirit to the demand inheritance rule. The difference is that no object is specified. Default rules can be used to implement open or closed world models.

A gacl essentially consists of a sequence of entries, where each entry is either a closure, inheritance, or default rule. Two kinds of gacl's can be defined. The key difference between them lies in conflict resolution. In an *ordered gacl*, conflict resolution is based on ordering. An *unordered gacl* does not incorporate a fixed conflict resolution algorithm as such. In an unordered gacl, all the entries are examined together to determine authorization. Incompleteness is *allowed* in both types of gacl's. If the entries in an ordered or unordered are inadequate to authorize or deny a request, then a "failure" is returned. An unordered gacl which results in contradictory authorizations returns the same.

A syntactic criterion restricts the class of policies that is supported. A policy is considered "well-formed" if it does not have recursive dependence in its closure and inheritance properties. This criterion simplifies the semantics but has a rather unfortunate consequence in that it limits expressiveness, for example it rules out such closure rules as, "a user who is allowed to write file F is also allowed to read F".

3.2 EACL

Another ACL-based language that incorporates conditional restriction on access rights is the Extended Access Control List language, or EACL, of Gheorghiu, Ryutov and Neuman [11]. The main idea behind EACL is to divide conditions into generic or specific. A *generic condition* is evaluated by the authorization mechanism. In contrast, a *specific condition* represents an application level check. Essentially, a specific condition is either evaluated by the application or an evaluation function specific to the condition must be passed along with the request. A specific condition helps make the corresponding application check explicit. Whether a condition is regarded as generic or specific depends on

the specific application where the EACL framework is used. For example, in the Prospero Resource Manager, time related conditions are considered generic, whereas resource consumption related conditions are considered specific. The EACL authorization model uses a flexible form of evaluation. A request is either authorized, denied, or *partially authorized.* A partially authorized request means the request is authorized pending certain specific-type conditions whose evaluation could not be carried out at the time of the request. In other words, the request would be authorized if these conditions were met.

ACI

Subjects can be hosts, users, groups, or applications. Objects can be hosts, files, printers, etc.

ACC

The access control context is not pre-defined but can include time.

ACR

An access control rule in EACL is specified with respect to subject, privilege, and condition. The condition component is optional and consists of a list of pairs of the form $< type : value >$, where

- *type* is a name of a specific or generic condition, and
- *value* is a restriction on the condition that must be satisfied in order to grant the request.

Similar to GACL, an operation is denied when there is a "-" preceding the operation name.

The policy evaluation algorithm is order based. The authorization entries already examined take precedence over later ones.

3.3 Comparison

The EACL language supports partial authorization decisions, which contrasts with the binary authorization model used in the database community. Note that GACL does not support partial authorization in the sense of EACL. The idea of partial authorization is originally found in PolicyMaker, a certificate authorization language which we will review in Section 6.1.

Table 8 compares GACL and EACL with two other more conventional OS authorization languages.

4 Temporal authorization languages

Usually, authorizations are specified with respect to the following dimensions: subject, object, access right, and conditional restriction on access right. Temporal authorization languages add another dimension to authorizations by their use of temporal constraints which allow the validity of authorization to be restricted to specific times.

Language	*Positive Auth.*	*Negative Auth.*	*Conditional restriction on access rights*	*Admin. Auth.*	*Flexible evaluation*
Multics [12]	Yes	Limited	No	Yes	No
Andrew [13]	Yes	Yes	No	Yes	No
GACL	Yes	Yes	Yes	No	No
EACL	Yes	Yes	Yes	No	Yes

Table 8: Comparison of OS authorization languages

4.1 Bertino, Bettini, Ferrari and Samarati

The authorization language of Bertino, Bettini, Ferrari and Samarati [14] (BBFS) allows temporal intervals to be associated with authorizations. Two kinds of temporal authorizations are supported. A *periodic temporal authorization* holds for periodic time intervals, for example the twenty-fifth of every month. In contrast, a *non-periodic temporal authorization* holds for a fixed time interval only, for example the working hours of 3/25/99. These two types of authorizations are built using periodic expressions. A periodic expression represents a set of periodic time intervals, possibly non-contiguous, for example the set of Mondays. The set of time intervals corresponding to a periodic expression P is formalized by the function $\Pi()$. If P is a periodic expression, *begin* is a time instant, and *end* is a time instant greater than or equal to the one denoted by *begin*, then the expression $< [begin, end], P >$ limits the application of P in the obvious way.

ACI

Subjects can be users. The set of objects and the set of access modes are left up to the policy writer.

ACC

The access control context includes time.

ACR

The basic form of access control rule is called *periodic temporal authorization.* A periodic temporal authorization is defined as a triple $([begin, end], P, auth)$, where $[begin, end]$ and P are as before and *auth* specifies an authorization with respect to subject, object, access right, authorization sign, and authorization issuer. If the authorization sign is the plus (minus) symbol, it indicates a positive (negative) authorization. The intuitive meaning is that *auth* is granted for each instant in $\Pi(P)$ that lies within $[begin, end]$. The other kind of rule, called *derivation rule*, can be used to specify temporal dependencies amongst authorizations. The derivation rule syntax is identical to that of period temporal authorization in all but the last argument. The last argument of a derivation rule is of the form $auth < OP > A$, where $< OP >$ is one of the operators WHENEVER, ASLONGAS, or UPON, and A is a boolean expression of authorizations which are of the same kind as *auth*. The choice of operators gives three different kinds of derivation rules.

- The WHENEVER rule is useful in situations where an authorization needs to be derived for each instant in $\Pi(P)$ at which a certain combination of authorizations is valid. For example, consider the following rule:

$$([95, \infty], \textit{Summer-time},$$
$$(\textit{summer-staff}, \textit{document}, read, +, \textit{manager})$$
$$WHENEVER$$
$$(\textit{regular-staff}, \textit{document}, read, +, \textit{manager}))$$

  This rule says that *summer-staff* has the authorization to read a certain *document* at a certain time in *Summer-time* if, at the same time, *regular-staff* have the authorization to read that document.

- The ASLONGAS rule is essentially a variant of the WHENEVER rule, with the following twist. If there exists even one instant in $\Pi(P)$ at which the combination of authorizations is not valid, then the derived authorization is blocked for all subsequent times in $\Pi(P)$.

- The dual case is covered by the UPON rule. If there exists even one instant in $\Pi(P)$ at which the combination of authorizations is valid, then the derived authorization is valid at all subsequent times in $\Pi(P)$.

Conflict resolution is based on the denials-take-precedence principle. In other words, a negative authorization wins over a positive authorization with the same subject, object, and access mode.

4.2 Atluri and Huang

Atluri and Huang [15] have proposed an authorization model for workflows, whose core is really a temporal authorization language (AH). A workflow is essentially a sequence of related tasks which are mutually dependent upon one another. One important aspect of workflow authorization is that subjects should be granted privileges only during the execution of a task and not any time before or after completion of the task. Usually, a workflow designer defines upper and lower bounds on the execution time of each task. However, the starting and ending times for a task can vary as long as the bounds are respected. Using authorizations with a predefined and fixed time interval for workflow authorization has the drawback that authorizations may become valid for a longer period than that actually desired. The AH language allows the granularity of authorized temporal intervals to be controlled dynamically.

ACI

Subjects can be users or roles.

ACC

The access control context includes time.

ACR

Two kinds of access control rules can be specified:

- A *task* definition allows to specify the lower and upper bounds on the time interval during which the task is allowed to be executed.
- An *authorization template* is specified with respect to subject, access right, and object.

More than one authorization template can be associated with each task. A built-in rule is used to derive authorizations from authorization templates and task definitions. A derived authorization can be thought of as an authorization having a dynamic time interval. The derivation rule can be described as follows.

1. If a request arrives within the pre-specified time interval but some time after the first authorized time instant, then the authorization is valid only from the time at which the request arrives.
2. If the task finishes ahead of the pre-specified ending time, then the derived authorization is revoked as soon as the task finishes.

Since only positive authorizations can be specified, this means revocation has to be done by deleting positive authorizations that exist in the authorization database. Notice this way of handling revocation seems rather simple-minded. A more elegant way is to allow positive and negative authorizations to co-exist.

4.3 Comparison

Table 9 compares the temporal authorization languages of Bertino *et al.* and Atluri and Huang.

Language	*Pos. and Neg. Auth.*	*Exceptions/ Strong Enforc.*	*Admin.*	*Temporal intervals*	*Subject grouping*
BBFS	Yes	No/No	No	static	users
AH	positive only	No/No	No	dynamic	roles

Table 9: Comparison of temporal authorization languages

5 Hypertext authorization languages

The hypertext paradigm facilitates access to a large number of information sources that are interlinked. Authorization languages for hypertext systems need to take into account different objects like documents, images and relationships amongst them.

5.1 Samarati, Bertino and Jajodia

One prominent language for hypertext authorization is that of Samarati, Bertino and Jajodia [16] (SBJ). It allows specification of authorizations with fine granularity for hypertext systems. The hypertext model upon which the language is based includes two kinds of objects. A *basic* object corresponds to concrete content such as the data contained in a .gif file. In contrast, a *node* object corresponds to a document schema; it contains information specifying the document's content. Nodes are divided into slots to allow specification of authorizations at different granularity levels. Links are elements that help relate various documents and basic objects. Two kinds of links are considered. An *inclusion link* states a relationship between a node and a basic object. It has the semantics that the content of a basic object is to be included in the document described by a node. A *navigation link* states a relationship between two nodes. It has the semantics that an appropriately authorized user can traverse the link to access the destination of the link. The source or destination of a link is called an *anchor*. A source anchor represents the starting point for a navigation or inclusion link. Analogously, a destination anchor represents the ending point. Note that by definition a basic object must appear as the destination anchor of an inclusion link. The destination anchor of a navigation link can be either a slot identifier or a node identifier. Users can traverse a navigation link by clicking a handle associated with it, which is basically some content that users perceive to be the link.

ACI

Subjects can be users working from specific remote locations (sites) or locally defined groups. Objects can be nodes, basic objects, groups of nodes, slots inside nodes, navigation links, and inclusion links. The notation "$\star@n$" means any user at location n. The wildcard "$\star$" when used as a subject means any user at any location and when used as an object means all slots in a node or any node, depending on the context.

ACR

The access control rules can be divided into four separate categories: viewing authorizations, navigation authorizations, authoring authorizations, and usage authorizations. These grant access respectively to view nodes or specific part of them, to navigate links, to edit node or link specifications, and to embed content of basic objects into documents. The different kinds of access modes supported allow, for example, to grant some users the right to modify a document, while restricting others to reading only.

A *viewing authorization* is specified with respect to a subject, node through which access is allowed, viewing right, and authorization object specified as a set of slots contained in one or more nodes. Four kinds of viewing rights are considered:

- The *view* right allows to see the specified document contents including the content retrieved through inclusion links. It is comparable to the traditional read right on files.

- The *view+* right basically means the same as view but additionally allows a user to see and traverse all navigation links located inside the document portions for which the user has view authorization. The success of the link traversal depends on the authorizations on the destination node of the link.

- The *view-all* right is used when the object of authorization is a directory. This right is similar in spirit to *view+* but restricts navigation to all nodes inside the specified directory for which the user has view authorization.

- The *view-all+* right is equivalent to *view+* on all the nodes in a specified directory.

The concept of *navigation authorization* decouples the right to activate navigation links from the right to specify them. The latter is controlled via *authoring authorizations.* A navigation authorization is specified with respect to a subject and navigation link. If the subject of a navigation authorization does not have view authorization on the node where the link appears, then the navigation authorization is considered ineffective. A clear reason for this is the fact that links are not visible as hypertext elements unless they are included in the document's content.

The concept of *usage authorization* decouples the right to use the content of basic objects from the right to specify inclusion links. The latter is controlled via authoring authorizations. A usage authorization is specified with respect to a subject, basic object, and node owned by the subject. It essentially defines the semantics of inclusion links. An inclusion link originating from the node is considered effective only when the the node owner has a usage authorization on the basic object corresponding to the link destination.

An authoring authorization is specified with respect to a subject, authoring right, and authorization object specified similarly as in a viewing authorization. Four kinds of authoring rights are considered: *annotate*, *link*, *append*, and *update*. The right to specify a navigation or inclusion link is based on ownership. The creator of a node is by default allowed to specify links originating from that node regardless of whether he has the necessary authorizations on the destinations of links. He can also authorize others to add links to nodes by granting them the *link* authoring right.

The concept of *domain* is introduced to allow grouping of nodes stored at possibly different sites. A node owned by a user can be included in a domain owned by another user, called the *domain administrator*. This makes possible for the domain administrator to specify authorizations on nodes included in his domain, independently of authorizations specified by the node owner. The authorizations that can be specified by the domain administrator are however restricted to browsing authorizations. A *post authorization* allows a user to export a node owned by him into some domain. A *take authorization* allows a domain administrator to import nodes to his domain.

5.2 Trellis

Trellis is a Petri Net based model of hypertext authoring and browsing, described in Stotts and Furuta [17]. The model does not provide a textual authorization language as such, but it offers browsing access control capabilities via Petri Net structures. (A paper by Murata [18] gives an excellent tutorial introduction to Petri Nets.) A Petri Net consists of a set of *places*, drawn as circles, and a set of *transitions*, drawn as bars. Directed arcs connect places and transitions. In Trellis, a hypertext is modeled using a Petri Net in which (1) document contents are mapped to places and (2) navigation links are mapped to transitions. Browsing control in Trellis is based on the idea of a *marked hypertext.* A marking essentially puts zero or more *tokens*, drawn as dots, inside each place in the net. A transition t can be fired if each place that is incident on t contains at least one token. After firing t, exactly one token is removed from the incident places and deposited in

each place that follows t. A token inside a place indicates that the associated content can be viewed; when a place is empty the content cannot be viewed. A marking essentially characterizes the class of allowed browsing patterns in a hypertext. Different markings characterize different classes of browsing patterns. The firing semantics of a transition provide a basic mechanism for browsing access control.

Many different kinds of browsing security policies can be enforced in Trellis. One example is a "consumption" policy that allows access to a document D as long as the total number of times it has been accessed already does not exceed a pre-set limit. For example let us assume that there is only one transition incident upon D. The desired form of policy can then be implemented by having a place with the required number of tokens inside it guard the transition incident upon D. After the tokens are exhausted, D can no longer be viewed. Another example is a temporal security policy that permits access to a document D only if some other document D' was already visited previously. This form of context sensitive policy can be easily implemented in terms of Petri Net structures.

In Trellis, it is up to the document author to structure the net in order to achieve the desired access controls. This requires knowledge of Petri Net semantics, so one issue is what assistance can be provided when the structuring needed to achieve the desired access effects is not obvious. A related issue is to what extent the desired structuring can be automated. Can policies be specified in some way other than specifying them directly in terms of Petri Net structures?

5.3 Comparison

Note that in SBJ only positive authorizations can be specified; all accesses that are not explicitly authorized are implicitly denied. The lack of negative authorizations simplifies the language design but makes it hard to express security policies that allow access to all but a few users. The consideration of negative authorizations is left as future work by the language designers. In Trellis, authorizations are not explicitly specified in the usual sense. A reachability analysis of a Petri-Net-based hypertext is needed to verify which nodes can or cannot be reached from particular initial markings. While SBJ supports authorizations at different levels of granularity, it provides rather limited support for expressing policies where access to a node may depend on the path followed in arriving to that node. This is because the navigation context in viewing authorizations is limited to paths of length one.

Table 10 gives a comparison.

Language	*Pos. and Neg. Auth.*	*Path-dependent Auth.*	*Auth. granularity*
SBJ	positive	limited	fine
Trellis	not appl	yes	coarse

Table 10: Comparison of hypertext authorization languages

6 Certificate authorization (trust establishment) languages

Certificate authorization languages facilitate policies that are based on credentials or trust relationships. Such policies are especially applicable to public key infrastructures. A credential based policy usually states what certificates are trusted, or under what conditions trust may be deferred. For example, a user A may accept certificates signed directly by someone he trusts, B, but not those signed by C even if B trusts C.

6.1 PolicyMaker language

The work of Blaze, Feigenbaum and Lacy [19], which proposed the PolicyMaker system, was the first to introduce the notion of a trust-management engine, in which policies, credentials, and trust relationships are all described using a common language.

ACI

Subjects can be either users or groups. A user is identified by its public key.

ACR

In PolicyMaker, access control rules are termed *assertions*. An assertion can be either signed or unsigned. Signed assertions are similar to credentials, which represent statement by others. Unsigned assertions are used as the ultimate source of authorization decisions. A signed or unsigned assertion is specified with respect to an issuer, the subjects to whom trust is delegated, and a predicate which specifies the class of actions delegated together with the conditions under which the delegation applies. The assertion syntax is:

Source ASSERTS AuthorityStruct WHERE Filter

where *Source* is the assertion issuer, *AuthorityStruct* is a set of subjects to whom trust is delegated, and *Filter* is a predicate. The Source entry can be either the reserved word POLICY or the public key of some entity. The former choice indicates an unsigned assertion. The latter choice indicates a signed assertion. The AuthorityStruct entry essentially represents a set of one or more public keys. This set can be specified explicitly or implicitly. The Filter entry corresponds to a program written in a language external to PolicyMaker. The choice of the filter language is up to the policy writer as long as it can be safely interpreted. In essence, this means programs written in the filter language are run within a restricted environment. Two classes of filters are considered. The first class either accepts or rejects action strings. The other class is more flexible. If an action string lacks information that would have otherwise allowed an assertion to succeed, then the filer emits the missing information and appends it to the original string. This kind of filter provides a basic mechanism for supporting partially authorized requests.

The policy evaluation algorithm used in the Policy Maker system is informally described in Blaze *et al* [19]. A related paper by Blaze, Feigenbaum and Strauss [20] treats it in considerable depth. The basic idea is to model a given set of assertions as a directed acyclic graph, whose vertices are labeled by public keys or policy sources and arcs are

labeled by filters. The set of arcs is determined as follows. If "v ASSERTS w WHERE f" is in the set of assertions, then there is an arc, labeled by f, from the vertex labeled by v to the vertex labeled by w. A query of the form "k REQUESTS ActionString" is satisfied when the digraph given by the assertions contains a chain $v_1 \xrightarrow{f_1} v_2 \xrightarrow{f_2} \cdots \xrightarrow{f_{t-1}} v_t$, where $v_1 =$ POLICY, $v_t = k$, and all the filters along the chain accept the ActionString.

6.2 KeyNote language

The KeyNote system has grown out of the earlier work on Policy Maker. The design philosophy behind KeyNote is discussed in a position paper by Blaze, Feigenbaum and Keromytis [21]. A full description of the KeyNote system appears as an Internet RFC [22].

The basic concepts underlying the KeyNote and PolicyMaker languages are similar. The programming unit for specifying policies is the same in both, namely assertions. The key difference between the two languages is that the definitions of predicates and actions strings are within the scope of the KeyNote language. Thus, KeyNote attempts to overcome two important inadequacies of PolicyMaker. Predicates in KeyNote assertions are specified using C-like expressions. Action strings are called *action attributes* in KeyNote and specified as name-value pairs. The KeyNote assertion syntax is more readable than PolicyMaker. An assertion can contain up to six fields:

- *Authorizer* field, which specifies the assertion issuer.
- *Comment* field.
- *Licensees* field, which specifies the principals authorized by the assertion.
- *Conditions* field, which specifies the conditions under which the Authorizer trusts the Licensees to perform an action.
- *Local-Constants* field, which specifies abbreviations which are made available for use in subsequent fields.
- *Signature* field, which specifies the encoded digital signature of the issuer over the assertion text.

All the fields above, except the Authorizer field, are optional. The *Authorizer*, *Conditions* and *Licensees* fields of KeyNote assertions have similar purpose to the *Source*, *Filter* and *AuthorityStruct* fields of PolicyMaker assertions.

6.3 PICSRules language

The PICSRules language [23] is a W3C standard for specifying filtering rules to allow or block access to Web pages. It was developed as part of a project titled Platform for Internet Content Selection, or PICS, in short. The PICS project aims to provide a set of standards for an open labeling platform supporting third party rating of Web content. The basic philosophy behind PICS is discussed in Resnik and Miller [24].

ACI

Subjects can be users wishing to access web pages using a suitable browser. Objects can be Web pages identified by their URLs. They can be specified using regular expression syntax. For example, the expression "http://www.grody.com:*/*" means any URL that specifies the host www.grody.com, regardless of the username, port number, or file path.

ACC

The access control context includes PICS labels that describe the ratings given to documents along one or more dimensions. Labels can be embedded in documents or provided by third parties, called rating services. Labels are retrieved from label bureaus. A PICSRules rule may specify what rating services and bureaus are used in the decision making. This information can thus be treated as part of the access control context.

ACR

The access control rules can be divided into two types, depending on whether they are specified with respect to URL patterns or label expressions. Rules of the former type accept or reject documents, based solely on their URLs. They pass or block a document depending on whether the requested URL matches the pattern specified in the rule. Two kinds of URL pattern based rules are provided. These are specified using the keywords *AcceptByURL* and *RejectByURL*. The AcceptByURL rule passes a document, whereas the RejectByURL blocks a document. Rules using labels pass or block a document depending on whether the label expression specified in the rule is satisfied. Four kinds of label expression based rules are provided. These are specified using the keywords *RejectIf*, *RejectUnless*, *AcceptIf* and *AcceptUnless*. The RejectIf and RejectUnless rules block a document, whereas the AcceptIf and AcceptUnless rules pass a document.

ACA

Rating services and label bureaus control and disseminate contents of labels, so they can be considered as part of the ACA.

A request is specified with respect to a URL and possibly any labels that are embedded in the document associated with the URL or retrieved from other sources. It is either authorized or denied. The rules are evaluated in sequential order. The first rule that matches with the request is used to decide whether the request is allowed or denied. If no rules are found, then access is allowed—that is, the open world model is used.

The RejectByURL, RejectIf and RejectUnless rules allow to specify negative authorizations explicitly. Since the number of Web documents that can be potentially viewed is generally huge, users often have security policies that give access to *all* except some documents. This observation justifies the design decision to support negative authorizations.

6.4 DL

Delegation Logic, or DL for short, is a certificate authorization recently proposed by Li, Feigenbaum and Grosof [25]. It is based on logic programs. Although the basic

philosophy behind DL is similar to PolicyMaker and KeyNote, it departs considerably in the choice of language syntax and semantics. DL has a much more abstract semantics than PolicyMaker and KeyNote.

The key features of DL are:

- As well as allowing statements that specify a decision in favor of authorizing, DL allows those specifying a decision against authorizing. DL uses negation to specify latter kind of statements. This facilitates policies that explicitly forbid something, which are termed *non-monotonic policies*. In contrast, PolicyMaker and KeyNote can only handle *monotonic policies*. An analogy with two high level policies for database authorization helps explain the difference. A monotonic policy is like the traditional closed world policy without negative authorizations. A non-monotonic policy is like the hybrid policy with positive as well as negative authorizations.

- In DL, delegation statements play the role that signed assertions play in PolicyMaker and KeyNote. An important difference is that in DL delegation statements specify an integer parameter, called *delegation depth*, which controls the extent to which delegations can be delegated further. For example, using a depth-2 delegation statement, a principal A can defer trust to another principal B and allow B to defer trust to still another principal C, but not allow C to defer it further. Probably, some kind of delegation depth control can be indirectly achieved by the use of filters in PolicyMaker. However, this is not a really clean way to handle it.

DL has two variations, D1LP and D2LP. The former supports only monotonic policies, whereas the latter supports non-monotonic policies.

ACI

Subjects can be users.

ACR

The access control rules in DL are built from statements. Two kinds of statements are considered: *delegation statements* and *direct statements*. In the former, trust is deferred whereas in the latter it is not. A direct statement essentially represents an assertion conveyed by a certificate; it expresses a statement made by someone. A direct statement is written "X *says* p", where X is a principal and p is a predicate.

A delegation statement is specified with respect to a principal, delegated predicate, delegation depth, and principal structure. It is written:

$$X \textit{ delegates } p\hat{}d \textit{ to } PS$$

where X is a principal, p is a delegated predicate, d is a delegation depth and PS is a principal structure. Here, X and PS play the same role as the Authorizer and Licensees fields in KeyNote assertions. DL allows variables to be used in the principal field, which adds to expressiveness. The delegated predicate corresponds to the class of actions authorized; it is decoupled from the conditions for validating authorizations themselves. This represents an improvement over the PolicyMaker and KeyNote assertion syntax, which mixes the

class of actions authorized with the conditions for validating authorizations, even though these concepts are essentially distinct. As an example, the delegation statement:

$$Alice\ delegates\ is\_key(\_,\_)\hat{}2\ to\ Bob$$

means Alice trusts Bob in making direct statements about the predicate *is_key*. The delegation depth 2 indicates that she also trusts anyone Bob trusts on the same predicate. Here, *is_key(_Key, _User)*, can for example mean that "_Key is a public key of user _User".

A principal structure essentially defines the sets of principals to whom a certain class of actions is delegated. These sets can be defined explicitly. Or, they can be defined implicitly via user defined predicates. Two kinds of principal structures are defined using the notion of a *principal-weight pair* set. A principal-weight pair set is simply a collection of pairs (A_i, w_i), where A_i is a principal and w_i is a positive integer, called weight. The principal structure, $threshold(k, \{(A_1, w_1), \ldots, (A_n, w_n)\})$, means sets S_i's such that $S_i \subseteq \{A_1, \ldots, A_n\}$ and the sum total of the weights for all principals in S_i is greater than or equal to k. Note that the definition of the above structure makes use of an explicitly specified principal-weight pair set. A more flexible structure, called *dynamic threshold structure*, allows this set to be implicitly defined. It is denoted $threshold(k, Prin\ says\ pred/x)$. Here, $pred/x$ means a user defined predicate *pred* of arity x. The arity can be either 1 or 2.

- The expression "*Prin says pred*/1" defines a principal-weight pair set whose principals are all principals A for which $Prin\ says\ pred(A)$ holds and the weight for each principal is 1.
- The expression "*Prin says pred*/2" defines a principal-weight pair set whose principals are all principals A for which $Prin\ says\ pred(A, w)$ holds for any positive integer w and the weight for each principal is the greatest positive integer w' which makes the corresponding expression true.

We recall below a concrete example from the DL paper in order to illustrate the use of threshold constructs. Suppose Bob, Sue, Carol, Joe and Peg are all Alice's friends who certify public keys. Alice trusts each of Bob, Sue, and Carol *fully* but trusts each of Joe and Peg only *partly*. She has the policy that a user must provide certificates issued by any two trusted friends or any one fully trusted friend. The following DL statements specify the required policy. Below the predicate $is\_key(\_Key, \_User)$ means the same as before.

```
Alice says fully_trusted(Bob).
Alice says fully_trusted(Sue).
Alice says partly_trusted(Carol).
Alice says partly_trusted(Joe).
Alice says partly_trusted(Peg).
Alice delegates is_key(_Key, _User)^1 to threshold(1, fully_trusted/1).
Alice delegates is_key(_Key, _User)^1 to threshold(2, partly_trusted/1).
```

An access control rule in DL is written

$$S\ if\ F,$$

where S is either a direct statement or a delegation statement and F is a statement expression formed by conjunctions and/or disjunctions.
The version of DL that supports non-monotonic policies is called D2LP. It extends D1LP statements with negation. Two kinds of negation are supported: classical and negation-as-failure. The first type is indicated by the symbol $\neg$ and the second type by $\sim$. If s is a statement, then $\neg s$ means s is known to be false, whereas $\sim s$ means s is *assumed* false in the absence of information to the contrary. The syntax for D2LP statements is:

A says $[\sim][\neg]p$ (direct statement)
A $[\sim][\neg]$ *delegates p^d to PS* (delegation statement)

where square brackets enclose things that are optional.

Each D2LP rule has an optional label attached to it. Labeling of rules provides a basis for resolving conflicts using prioritization information, which is specified via the predicate *overrides*/2. The fact *overrides*(*lab*1, *lab*2) means a rules with the label *lab*1 overrides a rule with label *lab*2. Another basis for conflict resolution is *recency*—that is, a more recently added rule overrides an older rule.

6.5 Comparison

Table 11 compares the general features of various certificate authorization languages. Note that in PolicyMaker, the languages for expressing action strings and filters are not fixed and left unspecified. This allows the expressiveness of general programs, but it also means the policy maker system as such is unsuitable for use by non-programmers.

Language	*Pos. and Neg. Auth.*	*Delegation depth control*	*Action language*	*Filter language*	*Flexible evaluation*
PolicyMaker	positive only	limited	not built-in	not built-in	Yes
KeyNote	positive only	limited	name-value pairs	C-like	No
PICSRules	Yes	limited		regular expr. and label expr.	No
DL	Yes	Yes	Prolog-like predicates	logic programs	No

Table 11: Comparison of certificate authorization languages

It is also worth comparing the kind of negation supported in DL with that in ASL and FAM (see Sections 2.3 and 2.4). In essence, only one kind of negation, negation-as-failure, is supported by ASL/FAM. They do not use classical negation to represent denial of authorization. Instead negative marks are used to capture denials.

A recent perspective on trust management is given in Blaze, Feigenbaum, Ioannidis and Keromytis [26].

7 Role-based authorization languages

7.1 Adage

Adage is a role based authorization language and toolkit developed at the Open Group Research Institute. The design philosophy behind Adage is discussed in Zurko, Simon and Sanfilippo [29]. It emphasizes usability features. One important feature of Adage is a graphical user interface that supports policy definition and maintenance by less advanced users.

ACI

Subjects can be roles and are called *actors*. Objects are called *targets*.

ACC

The Adage system uses a collection of volatile runtime information, called the *active state*, as one input to the access decision procedure. The active state includes information such as the number of principals active in a role (role cardinality) and the current labels of each principal (the active role set).

ACR

An access control rule is specified with respect to an actor, target and regulation. A *regulation* refers to an action as well as *constraints*—that is, conditions that allow or disallow the specified action. Thus, both positive and negative authorizations can be specified. Constraints can make comparisons between the attributes of some actor and target. The actor and target used can be the same as those with respect to which the rule containing the regulation is specified or can be different from them. More than one constraint can be associated with a regulation. Constraints can be combined using the logical operations AND, OR and NOT.

Templates can be defined for actors, targets, and regulations. They can be used in rules to refer to all entities of a particular type whose attributes match specified template values. A template for an actor, target, and regulation cannot specify attribute values other than the wildcard character for the principal, name, and action (respectively).

The above kind of rules essentially specify constraints on a role's action set once the role has been activated. These are called *non-activation rules*, as opposed to rules which specify constraints that are applied when a user is entering a role. The latter kind of rules, called *activation rules*, allow to specify constraints about the mutual exclusion of roles or cardinality of roles. Both *static* and *dynamic* constraints are supported. The difference between the two constraint types is that static constraints are enforced at policy definition time, whereas dynamic constraints require runtime support. Activation rules provide a basic support for two kinds of separation of duty policies, *static separation of duty policy* and *simple dynamic separation of duty policy*. In static separation of duty policy, no two roles can share the same users. In simple dynamic separation of duty, two roles can share the same users but a user cannot act in both roles simultaneously. The latter kind of policy is more flexible than the former but still not flexible enough in some applications. The more general and flexible version of separation of duty policy is known as *history-based*

separation of duty policy in the literature. (For a formal classification of various kinds of separation of duty policies, see Gligor, Gavrila and Ferraiolo [27].) Simon and Zurko [28] suggest several kinds of non-activation rules for representing history-based constraints in Adage. However, the Adage implementation as such does not support these rules. This is because of practical problems involved in representing and maintaining histories as "flat files".

Conflict resolution in Adage is based on the denials-take-precedence principle. A policy is specified by a set of rules and can be marked active or testing only. If a policy is marked active, it is used to enforce real authorization decisions. The testing only mode allows to debug policies without actual enforcement.

ACA

Access control administration in Adage is delegated to two special classes of users known as *project administrators* and *policy administrators*. A project administrator develops rules for a particular project. A policy administrator decides which users can act as project administrators and which policies are activated.

7.2 Chen and Sandhu

Chen and Sandhu [30] give a language for expressing role-based constraints, a core component of any role-based authorization language. They classify constraints as *system-level* or *application-level*. These are analogous to the role activation and non-activation constraints of Adage. The idea behind their language is to express constraints as restrictions on appropriately defined sets. While the use of set theoretic constructs allows greater expressiveness, it also assumes that users have the required mathematical knowledge.

ACI

Subjects can be roles.

ACC

The access control context includes externally defined functions that are used in specifying constraints.

ACR

The authorizations that can be specified are limited solely by the structure imposed on constraints.

The following pre-defined operators can be used in specifying constraints, amongst others:

- An operator that returns all roles which hold a given permission (action).
- An operator that returns all roles which a given user holds.
- An operator that returns all roles which are active in a given user session.

All of the above operators are referred to using a single identifier, *role-set*, by operator overloading. The dual operators are also defined, for example the operator *permission-set* returns all permissions for a given role when its argument is of type role.

Two non-deterministic functions are defined: *Oneelement* and *Allother*. The function Oneelement corresponds to extracting an element from a set. The function Allother corresponds to extracting the set obtained after deleting an element from a set. These are analogous to the Prolog predicates `member(X, List)` and `delete(X, List1, List2)`.

Constraints are written as formulas built using the following logical connectives: and ($\wedge$), or ($\vee$), not ($\neg$), and implication ($\rightarrow$). A variety of role-based constraints can be expressed, for example static separation of duty. If R is a set of mutually exclusive set of roles and U is a set of users, then static separation of duty is captured by the constraint:

$$\begin{aligned} & \mathit{oneelement}(R) \in \mathit{role\text{-}set}(\mathit{oneelement}(U)) \\ \rightarrow\ & \mathit{allother}(R) \cap \mathit{role\text{-}set}(\mathit{oneelement}(U)) = \phi \end{aligned}$$

The language seems quite rich yet it only partially accommodates history-based constraints. It allows externally defined functions to be referenced in specifying constraints, which is one way by which history-based constraints can be encoded. However, it is up to the policy writer to specify these functions.

7.3 Comparison

Table 12 compares Adage and CS. Neither of these support history-based SOD policies at the implementation level. Recently, Ahn and Sandhu [31] have given an abstract language for describing role-based separation of duty constraints, called RSL99, which essentially extends the previous work by Chen and Sandhu (CS). Again, their work does not address time or history.

Language	*Pos. and Neg. Auth.*	*Role Act. and Non-Act. rules*	*History-based SOD*
Adage	Yes	Yes	No
CS	positive only	Yes	No

Table 12: Comparison of role-based authorization languages

8 CSCW authorization languages

Authorization languages for collaborative systems can be distinguished from other kinds of languages by their use of *collaboration rights*. A collaboration right refers an operation whose result can affect multiple users. A large number of rights are necessary in collaborative systems.

8.1 Shen and Dewan

The authorization language of Shen and Dewan [32] is based on a generalized muti-user editing model of collaboration called Suite [33].

ACI

Subjects can be users or roles. Objects can be entities that are shared amongst different users. As an example, consider a program that is edited and tested by multiple users simultaneously. Each shared object is associated with a data structure, called *active variable*, which determines what object properties are shared amongst users.

ACR

An access control rule is specified with respect to a subject, object and access right. Both positive and negative rights are supported. Over 50 different rights are provided. The exact set of rights provided is motivated by the Suite framework. For example, the *ValueCoupledR* right allows a user to share value changes to a variable with others. As another example, the *DataR* right group includes several individual rights such as read and write rights. Note that a right group does not correspond to any specific operation on an object. It corresponds to a grouping of rights that is semantically meaningful.

Grouping can be used along each of the three authorization dimensions: subject, object and right. A fixed set of implication rules, called *inheritance rules*, determines how (implicit) authorizations are inferred from groups.

Object inheritance and conflict resolution: Active variables can be grouped, which allows authorization objects to be specified at different levels of granularity. If an object belongs to more than one group, then a mechanism called inheritance directive is used to decide how access rights are inherited. An inheritance directive is specified with respect to an object and right. It defines an ordered list of groups from which access rights can be inherited. If the rights inherited via these groups conflict, then the right inherited from a group that appears earlier in the list wins.

Right inheritance and conflict resolution: Two kinds of transitive relationships are defined over rights: *include* and *implies*. The include relationship essentially describes the right groups that are supported. If there exists an authorization stating that subject s is granted right group R on object o, then s inherits each right included in R on o. The include relationship is defined over individual rights as well as right groups.

The imply relationship is defined only over individual rights but not right groups. It allows a user to inherit the right to a less powerful operation if he already has the right to a more powerful operation. For example, the right *InsertR* implies the right *ReadR*. The main purpose behind introducing the imply relationship is rights consistency. Both positive and negative rights can be inherited via the include relationship. In contrast, only positive rights can be inherited via the imply relationship. This is justified by the fact that denial of a more powerful operation (e.g. write) should not result in the denial of a less powerful operation (e.g. read). If conflicting rights are inherited via the include and imply relationships, then the imply relationship wins.

Subject inheritance and conflict resolution: Two kinds of inheritance relationships amongst subjects are considered: *take* and *have*. The take relationship allows a subject to inherit the rights of its roles, whereas the have relationship allows a subject to inherit only a subset of the positive rights of another subject. In essence, the take relationship defines a role hierarchy. It determines which role is more specific. If the rights inherited from two different roles via the take relationship conflict, then the more specific role wins. If the roles with conflicting rights are unrelated, then the conflict is resolved by ordering. The entry that appears earlier in the access list wins. If a right inherited via the take relationship conflicts with a right inherited via the have relationship, then the take relationship wins.

The set of conflict resolution rules is not claimed to eliminate all potential conflicts that can arise. The design decision to tolerate possible conflicts is justified by the fact that checking for consistency can be costly when the set of access objects is large.

A fuller language, which supports a richer set of protection objects such as sessions and windows, is reported by the authors in an extended work [34]. Another paper by the same authors [35] describes a model of access administration which provides a flexible choice of policies for meta access-control.

8.2 Intermezzo language

The Intermezzo framework proposed by Edwards [36] allows to specify coordination policies for collaborative environments in terms of access control rights on data objects. Coordination policies describe how information that is useful in providing awareness about user activities in a collaborative environment should be regulated. They aim to limit the dynamism in collaboration but without discouraging potential interaction between participants. In contrast, application-specific policies describe how application-specific data, such as text in a shared editor, is to be regulated.

ACI

Subjects can be users or roles. Three kinds of roles are considered: *simple*, *dynamic* and *aggregate*. Membership of a simple role is pre-defined and static. In contrast, membership of a dynamic role is determined by a potentially arbitrary predicate function that is evaluated at runtime. An aggregate role is simply a grouping of one or more roles.

The protection objects are called *resources*. Each resource has one or more *attributes*. Attributes store data including links. Basically, links are pointers to resources. They provide a basic mechanism for resource aggregation. Four kinds of operations on objects are considered: reading an object, writing an object, removing (deleting) an object, and testing for the existence of an object. A special resource, called *Activity*, is used to model user activities. An Activity resource is defined as an aggregation of the following three resources:

- a *Subject resource*, which specifies user information,
- a *Verb resource*, which specifies the task or application being run, and
- an *Object resource*, which specifies the file on which the task is operating.

Activity resources give a "world view" of users and their activities.

ACR

An access control rule in Intermezzo essentially binds a role to a *policy structure.* A policy structure indicates what operations are allowed on resources and their attributes. For example, the policy clause "resource Verb = NONE" indicates that all operations on Verb resource are disallowed. If some role is assigned a policy structure containing the above clause, then the task being performed by any user acting in that role is protected information. Positive authorizations can be specified with respect to individual operations, whereas negative authorizations can only be specified with respect to all operations. Authorizations can be specified for a resource as a whole together with authorizations for individual attributes within the resource. Authorizations specified at the attribute level within a resource take precedence over the authorizations specified for the resource as a whole. The closed world model is used in respect of authorizations specified for attributes. If a policy structure does not explicitly grant access to a particular attribute, then access is denied. If a user is in two roles which are assigned policies with conflicting access rights, then the resulting conflict is resolved by having the policy that allows the operation override the policy that disallows the operation. The intention behind this rule is to solve conflicts in a liberal (unusual) way.

Anonymity and *pseudonymity* are two high level co-ordination policies supported by Intermezzo. Anonymity requires that information that can be used to identify users, either directly or indirectly, should be restricted. Pseudonymity is a less restrictive policy; it allows selective access to information such as tracking of individual activities. The former policy is specified by giving the NONE right to all attributes within the *Subject* and *Verb* resources. However, one can still allow determination of the existence of these resources as a whole by giving the EXIST right on these resources. The latter policy is specified by giving READ rights to the *Location* attribute within the *Subject* resource as well as to all attributes within the *Activity* resource.

8.3 Comparison

Table 13 compares the two languages reviewed in this section. In Intermezzo, negative authorizations can be specified only for all rights. Recall from Table 8 that this feature is also found in the access control lists used in the Multics operating system [12].

Language	*Pos. and Neg. Auth.*	*Rights grouping*	*Object aggregation*	*Subject grouping*	*Admin*
SD	Yes	Yes	Yes	roles	Yes
Intermezzo	limited	No	Yes	roles	No

Table 13: Comparison of CSCW authorization languages

References

[1] Handbook of Programming Languages Vol. 3: Little Languages and Tools (ed. P. Salus). MacMillan Computer Press, 1998.

[2] M. Abrams, K. Eggers, L. La Padula and I. Olson. A Generalized Framework for Access Control: An Informal Description. *Proc. 13th National Computer Security Conference*, pp. 135–143, 1990.

[3] P. Griffiths and B. Wade. An authorization mechanism for a relational database system. *ACM Trans Database Systems*, Vol. 1, No. 3, pp. 242–255, Sep. 1976.

[4] S. Jajodia, P. Samarati and V. Subrahmanian. A Logical Language for Expressing Authorizations. *Proc. 1997 IEEE Symposium on Security and Privacy*, pp. 31–42, May 1997.

[5] J. Ullman. Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press, 1988.

[6] E. Bertino, S. Jajodia and P. Samarati. Supporting Multiple Access Control Policies in Database Systems. *Proc. 1996 IEEE Symposium on Security and Privacy*, pp. 94–107, May 1996.

[7] F. Rabitti, E. Bertino, W. Kim and D. Woelk. A Model of Authorization for Next-Generation Database Systems. *ACM Trans Database Systems*, Vol. 16, No. 1, pp. 88–131, Mar 1991.

[8] D. Ferraiolo, J. Cugini and D. Kuhn. Role-Based Access Control (RBAC): Features and Motivations. *Proc. 1995 Computer Security Applications Conference*, pp. 241–248, Dec 1995.

[9] S. Jajodia, P. Samarati, V. Subrahmanian and E. Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 474–485, May 1997.

[10] T. Woo and S. Lam. Designing a Distributed Authorization Service. *Proc. IEEE INFOCOM'98*, pp. 419–429, Nov. 1998.

[11] G. Gheorghiu, T. Ryutov and B. Neuman. Authorization for Metacomputing Applications. *Proc. 7th IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[12] J. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, Vol. 17, No. 7, pp. 388–402, July 1974.

[13] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Trans Computer Systems*, Vol. 7, No. 3, pp. 247–280, Aug. 1989.

[14] E. Bertino, C. Bettini, E. Ferrari and P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Trans Database Systems*, Vol. 23, No. 3, pp. 231–285, Sep. 1998.

[15] V. Atluri and W.-K. Huang. An Authorization Model for Workflows. *Proc. European Symposium on Research in Computer Security, Lecture Notes in Computer Science, Vol. 1146*, pp. 44–64, Springer-Verlag, 1996.

[16] P. Samarati, E. Bertino and S. Jajodia. An Authorization Model for a Distributed Hypertext System. *IEEE Trans Knowledge and Data Engineering*, Vol. 8, No. 4, pp. 555–562, Aug. 1998.

[17] P. Stotts and R. Furuta. Access Control and Verification in Petri-Net-Based Hyperdocuments. *Proc. COMPASS'89*, pp. 49–55, 1989.

[18] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE*, Vol. 77, No. 4, pp. 541–580, Apr. 1989.

[19] M. Blaze, J. Feigenbaum and J. Lacy. Decentralized Trust Management. *Proc. 1996 IEEE Symposium on Security and Privacy*, pp. 164–173, May 1996.

[20] M. Blaze, J. Feigenbaum and M. Strauss. Compliance Checking in the Policy-Maker Trust-Management System. *Proc. of the Financial Cryptography '98, Lecture Notes in Computer Science, vol. 1465*, pp. 254–274, Springer-Verlag, 1998.

[21] M. Blaze, J. Feigenbaum and A. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. *Proc. 1998 Cambridge Security Protocols International Workshop, Lecture Notes in Computer Science, vol. 1550*, pp. 59–63, Springer-Verlag, 1999.

[22] M. Blaze, J. Feigenbaum, J. Ioannidis and A. Keromytis. The KeyNote Trust-Management System (Version 2). Internet RFC. http://www.cis.upenn.edu/ angelos/Papers/rfcnnnn.txt

[23] C. Evans, C. Feather, A. Hopmann, M. Presler and P. Resnick. PICSRules 1.1. W3C Recommendation, December 1997. http://www.w3.org/TR/REC-PICSRules.

[24] P. Resnik and J. Miller. PICS: Internet Access Controls Without Censorship. *Communications of the ACM*, Vol. 39, No. 10, pp. 87–93, 1996.

[25] N. Li, J. Feigenbaum and B. Grosof. A Logic-based Knowledge Representation for Authorization with Delegation (Extended Abstract). *Proc. 12th IEEE Computer Security Foundations Workshop*, pp. 162–174, July 1999.

[26] M. Blaze, J. Feigenbaum, J. Ioannidis and A. Keromytis. *Secure Internet Programming: Issues in Distributed and Mobile Object Systems, Lecture Notes in Computer Science State-of-the-Art Series, Vol. 1603*, pp. 185–210, Springer-Verlag, 1999.

[27] V. Gligor, S. Gavrila and D. Ferraiolo. On the Formal Definition of Separation-of-Duty Policies and their Composition. *Proc. 1998 IEEE Symposium on Security and Privacy*, pp. 172–183, May 1998.

[28] R. Simon and M. Zurko. Separation of Duty in Role-Based Environments. *Proc. 10th IEEE Computer Security Foundations Workshop*, pp. 183–194, June 1997.

[29] M. Zurko, R. Simon and T. Sanfilippo. A User-Centered, Modular Authorization Service Built on an RBAC Foundation. *Proc. 1999 IEEE Symposium on Security and Privacy*, May 1999.

[30] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. *Proc. 1st ACM Workshop on Role-Based Access Control*, pp. 39–46, 1995.

[31] G.-J. Ahn and R. Sandhu. The RSL99 Language for Role-Based Separation of Duty Constraints. *Proc. 4th ACM Workshop on Role-Based Access Control*, Nov. 1999, to appear.

[32] H. Shen and P. Dewan. Access Control for Collaborative Environments. *Proc. ACM 1992 Conference on Computer Supported Cooperative Work*, pp. 51–58, Nov. 1992.

[33] P. Dewan and R. Choudhary. A high-level and flexible framework for implementing multi-user interfaces. *ACM Trans Information Systems*, Vol. 10, No. 4, pp. 345–380, Oct 1992.

[34] P. Dewan and H. Shen. Controlling Access in Multiuser Interfaces. *ACM Trans Computer-Human Interaction*, Vol. 5, No. 1, pp. 34–62, Mar 1998.

[35] P. Dewan and H. Shen. Flexible Meta Access-Control for Collaborative Applications. *Proc. ACM 1998 Conference on Computer Supported Cooperative Work*, pp. , Nov. 1998.

[36] W. Edwards. Policies and Roles in Collaborative Applications. *Proc. ACM 1996 Conference on Computer Supported Cooperative Work*, pp. 11–20, Nov. 1996.