An approach to authorization modelling in object-oriented database systems

Bertino, E.; Weigand, H.

Publication date:
1991

Document Version
Publisher's PDF, also known as Version of record

[redacted]

ITK

RESEARCH
REPORT

ITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE

ITK Research Report
december 1991

An Approach to Authorization Modelling in Object-Oriented Database Systems

E. Bertino and H. Weigand

No. 32

ISSN 0924-7807

An Approach to Authorization Modeling in Object-Oriented Database Systems

E.Bertino (*)[1], H.Weigand (**)
(*) Dipartimento di Matematica - Universita' di Genova
Via L.B. Alberti 4, 16132 Genova (Italy)
e-mail: bertino@igecuniv.bitnet
(**) INFOLAB, Tilburg University
P.O. Box 90153
5000 LE Tilburg (The Netherlands)
e-mail: weigand@kub.nl

Abstract

Authorization is an important functionality that every data management system should provide. An authorization mechanism allows different access rights on data items to be selectively assigned to users. Authorization models and mechanisms have been widely investigated in the framework of traditional database systems. The extension of those models and mechanisms to advanced data management systems is quite complex, because those systems are characterized by data models with a larger number of semantic constructs than traditional models, like the relational one. A first authorization model defined for object-oriented (and semantic) database systems has been presented in [Rabi 91]. In this paper we present an authorization model that substantially extends and revises that model. The most significant extension concerns the support for content-dependent authorization, which was not provided in [Rabi 91]. Content-dependent authorization is very important in providing an authorization mechanism able to directly support authorization policies of application environments. Moreover, it is a crucial functionality in environment where data objects frequently change their status. In addition, the model presented here differs from the model defined in [Rabi 91] under several aspects that are pointed out in the paper.

1. Introduction

The technology of object-oriented database management systems (OODBMSs) is very promising to a number of applications in business, and industry. An OODBMS combines the features of object-oriented programming languages with those of database management systems [Bert 91]. Therefore, an OODBMS is a powerful environment supporting application design, development and evolution, providing at the same time functionalities ensuring data protection and security. In particular, preventing unauthorized accesses to data stored into the database is a functionality that is required by

[1] The work reported in this paper was carried out by E.Bertino when visiting the University of Tilburg during summer 1991.

most applications, both traditional and advanced [Jajo 91]. It is very common that an organization uses information with different degrees of sensitivity. Therefore, data management systems must provide capabilities to selectively share data among multiple users.

Although most commercial DBMSs have security subsystems supporting access control, authorization in object-oriented (and advanced) database systems has not yet been fully investigated. The definition of a suitable authorization model for those systems poses several requirements. First of all, the unit of authorization must be the object, since objects are the access units. However, the authorization model should support different levels of granularity for both performance and user convenience reasons. For example, it should be possible to grant authorization on a single object, but also on an entire class, or on an entire database. The model should take into account all semantic modeling constructs commonly found in object-oriented data models, such as composite objects and versions.

An object-oriented authorization model, satisfying the previous requirements, has been defined in the Orion system framework [Rabi 91]. Other systems implement less sophisticated models or have no authorization at all. The model defined in [Rabi 91] uses the object as the authorization unit. In addition, this authorization model accounts for semantic modeling aspects, such as inheritance hierarchy, versions, and composite objects. However, a limitation of the model defined in [Rabi 91] is that it does not support authorization that depends on object contents. A way of supporting this in relational databases is through views (see for example [Bert 88]). A view states conditions (expressed as the qualification clause of the view query) that the tuples must verify in order to be accessed by a user (or group of users). In this paper, we present an authorization model that extends the model of [Rabi 91] with content-dependent authorization rules, therefore providing the same function as protection views [Bert 88]. In addition, we simplify some aspects of the model defined in [Rabi 91] and we introduce two different modalities for authorization administration. Moreover, we outline how the model can be extended to support method-based authorization as well.

Recent work has been reported on mandatory authorization models for object-oriented databases and knowledge bases [Garv 91]. Although our model only deals with discretionary authorizations (as all commercial DBMSs do), it can be seen as complementary to mandatory authorization models. As pointed out in [Garv 91] within an access class [2] authorizations can be granted in a discretionary way, and therefore a model like the one defined in [Rabi 91] and here could be used to this purpose. Moreover, we will see that some conclusions developed in this paper share some ideas developed in [Garv 91]. This important because it shows that discretionary and mandatory authorization models share some fundamental properties, even though they are defined for different purposes. We will point out similarities within the paper. Finally, note that the model defined in [Garv 91] does not deal with content-dependent authorization.

The remainder of this paper is organized as follows. Section 2 presents an object-oriented data model that will be used as reference in describing the authorization model. Section 3 presents the authorization model. Section 4 discusses

---

[2] An access class is a concept of mandatory authorization models and it is defined as a pair (<sensitivity level, category set>) [Garv 91], where sensitivity level is ususually one among {TOP-SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED}, while a category represents a partition of the overall information space based on what the data deals with. Examples of categories could be: Navy, Airforce, Army. Therefore, the notion of access class should not be confused with the notion of class in object-oriented data models.

authorization administration; in particular, it presents different policies that are supported by the model in order to grant/revoke authorizations on objects. Finally, Section 5 presents some conclusions.

2. Preliminary Definitions

In this section, we first summarize the main features of object-oriented data models by a reference model which will be used in the rest of the paper for the discussion. This reference model should not be interpreted as a new model. Rather it is similar to the *core model* described in [Kim 90], in that has most features commonly found in various object-oriented data models. The reference model used in this paper also provides the notion of *composite objects* [Kim 89] and *versioned objects*, since the authorization model should account for these semantic modeling concepts as well. Then we present an overview of a constraint language based on [Wier 91]. This language will be used to express content-dependent authorization rules on both objects and subjects within the authorization model.

2.1 A Reference Object-Oriented Data Model

In this model a class is defined by specifying its name, its attributes, and the names of its superclass(es). Multiple inheritance and the existence of a default class, called TOP_CLASS, root of an inheritance hierarchy encompassing the entire database are assumed. An attribute is defined by specifying its name and its domain. Classes have both the intensional and extensional meaning and an object can be *instance* of only one class. An object, however, can be *member* of several classes through the inheritance hierarchy. Attributes can be single-valued or multi-valued. In defining multi-valued attributes, the various object-oriented data models use different constructors such as set, list, tree, array. In the reference model we will abstract from specific constructors, and we assume that multi-valued attributes are defined by using a constructor denoted as **set-of**. The following definitions specify a notation for the Reference Model.

If a_i is an attribute name and C_i is a class name then:
(1) $A_i = a_i : C_i$ is the definition of a single-valued attribute;
(2) $A_i = a_i :$ **set-of** C_i is the definition of a multi-valued attribute.

A method definition consists of a *signature* and a *body*. The signature specifies the method name, and the classes of the objects that are input and output parameters for the method. The body provides the implementation of the method and consists in a sequence of statements written in some programming language. If M is a method name, In_i $(1 \leq i \leq n)$ is an input parameter specification and Out is an output parameter specification, $M(In_1, In_2, ..., In_n) \rightarrow Out$ is a method signature definition. An input parameter specification consists of the parameter name and of the parameter domain. The parameter domain is a class name or can be defined as a collection of instances of a class, in the same manner as attributes are specified. An output parameter is a class name, or a collection of instances of a class. The invocation of a method M on an object O has the form $O.M(O_1, O_2,....,O_n)$ where $O_1, O_2,....,O_n$ are objects that are passed as input parameters.

Classes are recursively defined as follows:

- Integers, floats, strings, text, and Boolean are classes (called primitive classes)

- There is a special class, called TOP_CLASS, which has no superclass; it is default for superclass, if no superclasses are specified
- If $A_1, A_2, \ldots, A_n$ $(n \geq 1)$ are attribute definitions, with distinct names;
  if $M_1, M_2, \ldots, M_k$ $(k \geq 0)$ are method definitions, with distinct names;
  and $C, C_1, C_2, \ldots, C_h$ $(h \geq 0)$ are distinct class names; then

  Class C

  Attributes $A_1; A_2; \ldots; A_n;$

  Methods $M_1; M_2; \ldots; M_k;$

  Superclasses $C, C_1, C_2, \ldots, C_h$

  End

  is a class.

We assume that system-defined methods are provided (called *accessor methods*) that allow direct reading/writing of object attributes. These methods, called *Read* and *Write*, are used to handle the authorizations to read/write an object's attributes in a consistent way with the overall authorization model. These methods do not need to be implemented as general methods; they are implemented by the system in an efficient way. The semantics of the read is to simply return the value of an attribute, while the write simply updates the attribute by assigning to it a new value. In the following, given an object O and an attribute A_i, the notation $O.A_i$ will denote $O.Read(A_i)$. Similar methods are provided for reading and modifying a class definition (*Read-class, Modify-class*). In addition, methods are provided for creating and deleting objects, and classes (respectively *Create, Delete, Define-class, Delete-class*), and for adding, removing elements from multivalued attributes (*Insert, Remove*).

Composite objects

For composite objects we will use the same model as [Kim 89], since this model is quite general. The model defined in [Kim 89] distinguishes between two types of references among objects: *general*, and *composite*. The latter is used to model the fact that a referenced object is actually a part of (or component) of a given object. An object and all its components constitute a *composite object*. A similar model is also used in [Weig 91]. Moreover, in [Kim 89], composite references are further refined into *shared/exclusive* and *dependent/independent*. A shared composite reference allows the referenced object to be shared among several composite objects, while an exclusive composite reference constrains an object to be component of at most one composite object at the time. A dependent (independent) composite reference models the fact that a component object is dependent (independent) on the existence of the composite object(s) of which is part. These two dimensions can be orthogonally combined. Therefore, four different types of composite references are possible. Details can be found in [Kim 89]. In our reference model, composite references are specified using some special keyword in attribute definitions. If A_i is an attribute definition, composite attributes are specified as follows:

A_i **composite [shared | exclusive] [dependent | independent].**

The notion of composite object, in addition to being important from the semantic point of view, is important from the performance point of view. For example, the components of a composite object are very often clustered with the root of the composite object. Moreover, composite objects can be used as the unit of authorization.

Versioned objects

Several versioning models have been proposed in the literature; a survey can be found in [Katz 90]. Here, we present some basic aspects of versioning mechanisms that should be sufficient for discussing the authorization model. In general, a *versioned object* can be seen as a hierarchy of objects, called *version hierarchy*. Each object in a version hierarchy (except for the root object) is derived from another object in the hierarchy by changing the values of one or more attributes of the latter object. Objects in a version hierarchy are first-class objects. Therefore, they have their own object-identifier (OIDs). Information about the version hierarchy are often stored as part of the root object, called *generic object*. Two types of object references are supported in most version models to denote objects within a version hierarchy. The first is called *dynamic reference* and it is a reference to the generic version of a version hierarchy. It is used by users who do not require any specific version. The system selects a version (*default version*) to return to users. The default version is in most cases the most recent stable version. Commands are usually available that allow users to change the default version. The second type of reference is called *static* and it is a reference to a specific version within the version hierarchy. Another important aspect concerns stability of versions in version hierarchies. In most cases, versioned objects are shared among several users. Mechanisms should be provided so that users receive consistent and stable versions. Most version models distinguish between *transient* and *stable* versions. A transient version can be modified or deleted. However, no versions can be derived from a transient version. A transient version must first be *promoted* to a stable version before new versions can be derived from it. By contrast, a stable version cannot be modified. However, it can be used to generate new versions.

An illustrative example

An example of class definitions is presented in Figure 1.

```
Class Document
Attributes
        title: string;
        authorlist: set-of Employee;
        abstract: Paragraph composite exclusive dependent;
        content: set-of Section composite shared dependent;
        status: string;
        project: Project;
        status: string;
Methods
        Copy () → Document
End

Class Section
Attributes
        title: string;
        section_authors: set-of Employee;
        content: set-of Paragraph;
        date: string;
End

Class Paragraph
Attributes
        content: text;
        date: string;
End

Class Project
Attributes
        research_programme: string;
        manager: Employee;
End
```

Figure 1. Examples of class definitions

2.2 Constraint Language

The constraint language L is based on [Wier 91] of which we only use the static part. L is a simple first-order language with the following syntax.

- Examples of variables are p, b, ... There are infinitely many variables.
- Constants are *A101*, *1234* etc. There are infinitely many constants.
- There are finitely many function symbols, with metavariables f, ...
- There are finitely many predicate symbols, and the lettes P,Q,R are used as metavariables over predicate symbols. Each predicate symbol has an arity > 0. Two special predicates are the binary symmetric predicate = (equality), the binary predicate ≤ (subtype - to be defined below) and the binary predicate ∈ (membership - to be defined below).

Terms and formulas are built in the usual way using ∧,∨,¬,⇒,∀,∃, and punctuation symbols (,),[, and]. We use infix notation for =, ≤ and ∈. Below, we will add one new term type "path".

Let *Cl* be a finite set of constants. The elements of *Cl* are called class names and τ is used as a metavariable over *Cl*. The predicate ≤ defines a partial ordering on *Cl*. The constraint language L is extended in the following way.

- L contains a special binary predicate *class* and a set *Cl* of class names. The only well-formed formulas that can be built with *class* are of the form *class(t,τ)* for a term t and a class name τ.
- We introduce the abbrevations

  $\forall x{:}\tau\,(\phi(x)) \;:=\; \forall x\,(class(x,\tau) \Rightarrow \phi(x))$

  $\exists x{:}\tau\,(\phi(x)) \;:=\; \exists x\,(class(x,\tau) \Rightarrow \phi(x))$
- L contains two special ternary predicate symbols: *ATT*, where the first and third argument is always a class name (constant or variable), and the second argument an attribute name; and *ARITY*, where the first argument is a class name, the second argument an attribute name, and the third argument an arity constant from the set $\{1,n\}$. The set A is a finite set of constants called attribute names. We also introduce a ternary predicates *VAL* where the second argument is always an attribute name. In this way, we simulate a kind of second-order logic where we can quantify over attributes.

Examples of well-formed formulas with *ATT* and *VAL*:

ATT(document,title,string)

ATT(document,authorlist,employee)

VAL(c1,title,"an approach to authorization modelling")

VAL(c1,authorlist, elisa)

- Attributes of arity *1* are called single-valued, with arity *n* set-valued. We can use the definitions:

  $ATT1(t,a,z) \quad \Leftrightarrow \quad ATT(t,a,z) \wedge ARITY(t,a,1)$

  $ATTn(t,a,z) \quad \Leftrightarrow \quad ATT(t,a,z) \wedge ARITY(t,a,n)$

  $a \in Attset(t) \quad \Leftrightarrow \quad \exists y\,(ATT(t,a,y))$

- Attributes are inherited from superclass to subclass:

  $\forall t1,t2:class, \forall a:attribute ((a \in Attset(t2) \wedge t1 \leq t2) \Rightarrow a \in Attset(t1))$

- We isolate a special group of attributes called aggregation attributes, denoted as Agg. $Agg \subseteq A$. There is one special aggregation attribute *CONTENT*, and any aggregation attribute is a kind of content attribute in the following way:

  $\forall t1,z:class, \forall a:attribute (ATT(x,a,z) \wedge a \in Agg) \Rightarrow ATT(x,CONTENT,z))$

  $\forall t1,z:class, \forall a:attribute (VAL(x,a,z) \wedge a \in Agg) \Rightarrow VAL(x,CONTENT,z))$

We define paths in the following way:

- if x is a variable or constant (that is, an atom), and a an attribute name constant, then $x.a$ is a (basic) path.
- if p is a path and a an attribute name constant, then $p.a$ is a path

We say that a path p is single-valued iff all attributes are single-valued; otherwise the path is set-valued.

Path expressions can occur in equality and membership relations only. First, for basic paths:

$x.a = v$	$\Leftrightarrow$	$v = x.a$	$\Leftrightarrow$	$VAL(x,a,v)$	for single-valued attributes
$v \in x.a$	$\Leftrightarrow$	$VAL(x,a,v)$			for set-valued attributes

where v is a constant or variable. Instead of $VAL(x,CONTENT,v)$ we simply write $v \in x$.

The same, recursively, for single-valued paths p of length >1:

$$p.a = v \quad \Leftrightarrow \quad \exists c\,(p = c \wedge c.a = v)$$

where c is a "new" variable symbol not occurring already in the context of the expression.

For set-valued paths:

$v \in p.a$	$\Leftrightarrow$	$\exists c\,(c \in p \wedge VAL(c,a,v))$	when p is a path
$v \in p.a$	$\Leftrightarrow$	$VAL(p,a,v)$	when p is an atom (as above)

For example, the expression $y \in x.authorlist.name$ is equivalent to:

$$\exists c\,(c \in x.authorlist \wedge VAL(c,name,y)) \Leftrightarrow$$

$$\exists c\,(VAL(x,authorlist,c) \wedge VAL(c,name,y))$$

An example of a constraint in L is the following:

$\forall e:Employee, \forall d:Document\,(e \in d.authorlist \Rightarrow \exists m:Manager(e.manager = m \wedge m \in d.authorlist))$

which says that if an employee is on the authorlist of a document, then his manager is on the authorlist as well.

The second part of this formula can be simplified if we use the expression $p.b \in d.a$ as an abbrevation of:

$$\exists m(p.b = m \wedge m \in d.a)$$

(so note that the truth-value is false when $p.b$ does not exist).

The example then becomes $\forall e:Employee, \forall d:Document\,(e \in d.authorlist \Rightarrow e.manager \in d.authorlist))$

3. Authorization Model

The authorization model provides both *content-independent* and *content-dependent* authorization rules. According to [Fern 81], content-independent rules are authorization rules whose enforcement does not require accessing the object themselves;

whereas, the enforcement of content-dependent authorization rules requires accessing the objects. It is important to note that an authorization model should provide both types of authorization rules to provide adequate flexibility to the applications. In the remainder of this section, we first provide an overview of the authorization model, with special emphasis on content-dependent authorizations. Then, we present implication rules for subjects, objects, and authorization types. These rules are used here, as in [Rabi 91], to formalize the concept of implicit authorization. Implicit authorizations are those that can be derived from other explicit authorizations. Finally, we discuss content-dependent and content-independent authorizations for additional modeling concepts such as composite objects, versions, inheritance hierarchies.

3.1 Overview of the Authorization Model

Given S, O, and A being respectively the set of subjects in the system, the set of objects in the system, and a set of authorization types (that is, the authorization domains), an *explicit authorization* [Fern 81] consists of a predicate $b\_auth(s, o, a)$, where $s \in S, o \in O,$ *and* $a \in A$. The set of all explicit authorizations is called *Explicit Authorization Base* (shortly *EAB*). An example of explicit authorizations is $b\_auth(u_i, d[1], Read)$, denoting that the user u_i can read the document identified by $d[1]$.

A predicate *auth* is defined to determine if an authorization (s, o, a) is True or False. Given a triplet (s, o, a), if $auth(s, o, a)$ is True, then the subject s has an authorization of type a on object o. Therefore, the function of enforcing authorization can be simply seen as proving a goal in a logic program. In particular, we have that

$$\forall s: S, \forall o: O, \forall a: A\ (b\_auth(s,o,a) \Rightarrow auth(s,o,a)) \ (*)$$

that is, a subject s has an authorization of type a on an object o if an explicit authorization exists. In the remainder, we will use the predicate *auth* to denote both explicit and implicit authorizations, unless there is the need of distinguishing them. Moreover, we assume that when the predicate $auth(s,o,a)$ cannot be proved to be True, the authorization function will return False. That is consistent with the approach taken by most systems. Note that the formalism we will use within the paper to define authorizations is not meant to be the language used by the end-users. Rather, it can be used as a language for the implementation of authorization functions, since it can be easily mapped to a logical language, like Prolog. In Appendix A, we sketch a SQL-like user language for authorization.

In general, a simple way to handle authorization is to explicitly list all authorizations as explicit authorizations. However, this approach does not take advantage of the fact that an authorization may imply other authorizations. Therefore, there would be no need of inserting into the *EAB* the latter; rather, a better approach is to establish rules that allows the latter to be derived from the former. An example of derived authorization is that a manager should be able to access any information that his employees may access. The set of rules used to derive authorizations is called *Intensional Authorization Base* (shorthly *IAB*). The *Authorization Base* (shortly *AB*) then consist of *EAB* and of *IAB*. Note that derivation rules can be specified along all the three domains of authorization [Rabi 91].

However, the derivation rules stated in [Rabi 91] are independent of the content of objects and cannot contain user-specified predicates. An example of user-specified predicate is "An employee can read a document only if the employee is in the list

of authors of the document". Therefore, enforcing this authorization requires looking at the content of the attribute 'authorlist' of the given document. A simple way to formalize those predicates is by using the constraint language outlined in the previous Section. The above example would be formalized by the following rule:
$\forall e$: *Employee*, $\forall d$: *Document* ($e \in d.authorlist \Rightarrow auth(e,d,Read)$).

We note that content-dependent authorization rules may interfere with content-independent authorization rules. For example, suppose that a user u has received a Read authorization on document $d[i]$ from another user (that is, $b\_auth(u, d[i], Read)$ is True), and suppose that u is not among the list of authors of $d[i]$. Then the question is whether u should be allowed to read the document or not. The answer is that the u is allowed. In fact, $auth(u,d[i],Read)$ follows directly from $b\_auth(u,d[i], Read)$ (because of the basic implication (*)). Therefore when both content-independent and content-dependent authorization rules exist on an object, a subject may access the object if he verifies at least one of the authorization rules. In the above example, user u may read the document if either he has been explicitly authorized to reading it (even if he is not in the author list) or if he is one of the authors (even if he has not been explicitly authorized).

Therefore, the basic policy of this authorization model is that a user is allowed to perform a given action if he has either a content-independent or a content-dependent authorization to do so. Note that this is in accord with authorization policies followed in most commercial DBMSs. For example, in SQL/DS [SQL81], a user may receive multiple independent authorizations on the same relation. Moreover, he may also receive a content-dependent authorization on a relation if he is granted a privilege on a view defined on this relation.

Content-dependent authorizations can be explicit or implicit. An explicit content-dependent authorization has the same format of a content-independent authorization and in addition it has a predicate, stating the conditions under which the subject is authorized to access a given object. Some examples of content-dependent implicit authorization rules will be presented later on in the paper.

3.2 Inference Rules for Authorization Subjects

As in [Rabi 91] and [Wier 91], we organize users into *roles*. A given user may belong to several roles (or groups). Roles are often used in authorization models, since they allow a single authorization to be used to assign a privilege to a set of users, without having to grant specific authorization to each one of those users. In the framework of this authorization model, we consider the set of all roles to be a partially ordered set R called *role graph*. Because of the ordering among the nodes of the role graph, we have the following property holding for roles:

given two roles r_i and r_k such that $r_i \leq r_k$, $\forall s : S\ (s \in r_i \Rightarrow s \in r_k)$

that is, if a user belongs to a role, he belongs to all the super-roles of this role. We call this implication rules for roles (briefly this rule is denoted as I_r). This has an important consequence on authorization, because all authorizations that are valid on a role are then inherited by its sub-roles. This can be easily shown by observing the following:

1) an authorization of type a on an object o being valid for a role means that belonging to this role is a sufficient condition for holding this authorization: $\forall s : S\ (s \in r_k \Rightarrow auth(s,o,a))$
2) a user belonging to a role also belongs to all its super-roles (for implication rule I_r), that is, given $r_i \leq r_k$

$$\forall s : S\ (s \in r_i \Rightarrow s \in r_k).$$

By combining (1) and (2) above, we obtain that $\forall s : S\ (s \in r_i \Rightarrow auth(s,o,a))$.

Given an object o and an authorization a, the abbreviation $auth(r_k,o,a)$ will denote $\forall s : S\ (s \in r_k \Rightarrow auth(s,o,a))$.

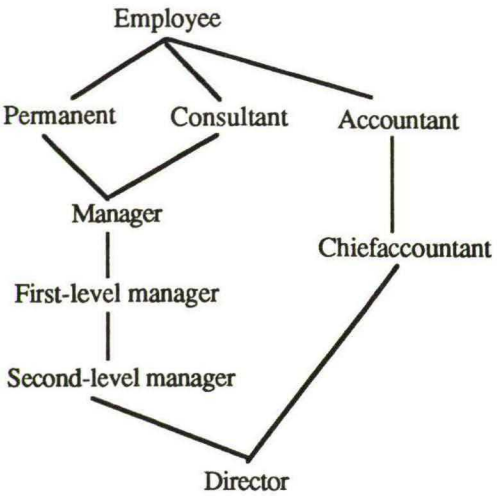

Figure 2. Example of Role Graph

Note that this implication rule for roles can be used also for content-dependent access rules. As an example, consider the role graph of Figure 2 and the authorization rule:

$\forall e$: *Employee*, $\forall d$: *Document* $(e \in d.authorlist \Rightarrow auth(e,d,Read))$ (a).

By using rule I_r, we obtain that also the following authorization holds:

$\forall m$: *Manager*, $\forall d$: *Document* $(m \in d.authorlist \Rightarrow auth(m,d,Read))$ (b).

Therefore, sub-roles inherit all authorization rules held by their super-roles. This is in accord with the results obtained in [Wier 91] for the inheritance of constraints along type hierarchies (in particular the results holding for sufficient conditions of integrity constraints). It is possible for a role to receive multiple authorizations concerning the same authorization type for the same object. These multiple authorizations may arise either because of multiple inheritance (a role may have several super-roles), or just because explicit authorizations are directly granted to this role. As an example, suppose that the following authorization policy is defined:

1) an employee may read a document if the employee is in the author list of the document

2) a manager can read all documents of the project he is managing.

The formalization of the first authorization rule has been shown before (authorization rule (a)). By using implication rule I_r, we obtain that a manager can access a document if the manager is in the author list of the document (derived authorization rule (b)). This is an example of an implicit content-dependent authorization. The second authorization rule is expressed as follows:

$\forall m$: *Manager*, $\forall d$: *Document* $(m = d.project.manager \Rightarrow auth(m,d,Read))$ (c).

Therefore the overall authorization rule for manager is obtained as an authorization rule having a sufficient condition obtained as the disjunction of the sufficient conditions of rules (b) and (c) above. Therefore:

$\forall m$: *Manager*, $\forall d$: *Document* $((m = d.project.manager \vee m \in d.authorlist) \Rightarrow auth(m,d,Read))$.

We model user roles, and authorization subjects in general, as classes using the model defined in the previous section. The only difference is that all classes concerning roles belong to an inheritance hierarchy rooted at a special class, called *User*. Modelling the authorization subjects as classes is very useful from several point of views. First, we can use those classes as domains of other classes, and therefore we can define authorization constraints involving both authorization subjects and authorization objects. Moreover, it is possible to define additional attributes describing users and roles within the system. This is important since it allows to add application-dependent information about users that can be pertinent to authorizations. As results, extensibility is achieved.

3.2 Authorization Types

Another dimension for deducing authorizations is represented by the set of authorization types. For example, it would be useful to model the fact that the authorization to write an object implies the authorization to read the same object. This is very useful especially when, as in the authorization model we are defining and in the one presented in [Rabi 91], authorizations can be granted on a single object and, therefore, there could be a large number of authorizations. The usage of implication rules for authorization types allows a single authorization on a object to be granted to a user and then all other authorizations to be derived implicitly from it.

Before specifying implication rules for authorization types, we need to define which are the possible authorization types. These are clearly related to the possible operations (that is, methods) defined for objects in the model. Note that we restrict our discussion to only system-defined methods (cf. Section 2). We will discuss authorization for user-defined methods in Section 3.4. Moreover, authorization types concerning composite objects and versions will be discussed in Section 3.4.

Table I presents the possible authorization types. For each authorization type, Table I specifies the system-defined methods whose executions are enabled by the authorization type. Note from Table I that there are some authorization types that do not correspond directly to some system-defined methods. However, they are used as shorthand for denoting a set of authorization types. For example, the *Read* authorization type is used to grant the possibility of reading any attribute of a given object. Similarly, the *Read-all* authorization type is used to grant the possibility of reading all instances of a given class, that is, to read all attributes of all instances of a given class. Note that among the authorization types, there are *Read*(A_i)*-all*; and *Write*(A_i)*-all*. Those authorization types are actually "parametric" ones, since they depend on the attributes specified in the class. The meaning of this authorization type is to allow a given subject to read (write) attribute A_i for all instances of the class. An example of the usage of this authorization type is the following:

b_auth(Employee, Document, Read(abstract)-all)[3]

---

[3] For content-independent authorizations (both implicit and explicit) we omit the universal quantifications; so this example is a shorthand for $\forall$ *e: Employee*, $\forall$ *d: Document (b_auth(e, d, Read(Abstract)-all))*.

This authorization states that all employees can read the abstracts of all documents. Note that in expressing the above authorization we used the shorthand previously defined for implication rules among subjects. Moreover, note that expressing the formal semantics of the above authorization rule, (that is, the fact that it implies the authorization type *Read(Abstract)* for all instances of class *Document*) we have to wait after we introduce the implication rules for authorization objects (Subsection 3.3).

	Authorization type	Actions allowed
Database	Read	Read the database definition
	Read-all	Read all objects within the database
	Write-all	Modify all objects within the database
	Create	Create a class
Class	Read	Read class definition
	Write	Modify class definition
	Delete	Delete a class
	Read-all	Read all instances of the class
	Write-all	Update all instances of the class
	Read(Ai)-all	Read attribute Ai for all instances of the class
	Write(Ai)-all	Update attribute Ai for all instances of the class
	Create	Create new instances of the class
Instance	Read	Read all attributes of the instance
	Write	Modify all attributes of the instance
	Delete	Delete the instance
	Read(Ai)	Read attribute Ai of the instance
	Write(Ai)	Modify attribute Ai of the instance; also insert, and remove if Ai is multivalued

Table I. Authorization Types

In defining the implication rules for authorization types, we make the assumption that the set O of all objects is partitioned into three sets: *Database* (the set of all databases), *Class* (the set of all classes), *Instance* (the set of all instances of some classes). Note that in the present model, we do not have meta-classes. If we had meta-classes, we should have added a fourth set. The following implication rules define implicit authorizations for authorization types. We first list the implication rules that are common to both classes and instances. Then we present those that are specific to instances and to classes.

I_{O1} $\forall s: S, \forall O_i : Class \cup Instance \ (auth(s, O_i, Write) \Rightarrow auth(s, O_i, Read))$

I_{O2} $\forall s: S, \forall O_i : Class \cup Instance \ (auth(s, O_i, Delete) \Rightarrow auth(s, O_i, Read))$

Additional implication rules for classes are as follows:

I_{C1} $\forall s: S, \forall C_i : Class \ (auth(s, C_i, Read\text{-}all) \Rightarrow auth(s, C_i, Read))$

This rule states that if a subject can read all instances of a class, then he can read also the class definition. This implication is quite reasonable, since in order to send messages to instances, a user must know their definition. Therefore, it does not make sense granting a user the read authorization on all instances of a class and not granting

the authorization to read the class definition. A similar conclusion has been reached in [Garv 91], where it is stated that the security level of an object is higher than the level of its definition.

I_{C2} $\forall s: S, \forall C_i : Class\ (auth(s, C_i, Write\text{-}all) \Rightarrow auth(s, C_i, Read\text{-}all))$

Note that because of rule I_{C1}, a user receiving the authorization to write all instances of a class, also implicitly receives the authorization to read the class definition.

I_{C3} $\forall s: S, \forall C_i : Class, \forall A_i : Att\text{-}set(C_i)\ \ auth(s, C_i, Write\text{-}all(A_i)) \Rightarrow auth(s, C_i, Read\text{-}all(A_i)))$

I_{C4} $\forall s: S, \forall C_i : Class, \forall A_i : Att\text{-}set(C_i)\ \ auth(s, C_i, Write\text{-}all) \Rightarrow auth(s, C_i, Write\text{-}all(A_i)))$

I_{C5} $\forall s: S, \forall C_i : Class, \forall A_i : Att\text{-}set(C_i)\ \ auth(s, C_i, Read\text{-}all) \Rightarrow auth(s, C_i, Read\text{-}all(A_i)))$

I_{C6} $\forall s: S, \forall C_i : Class\ (auth(s, C_i, Create) \Rightarrow auth(s, C_i, Read))$

As for implication rule I_{C1}, if a user can create instances, he must be able to read the class definition.

Note from the above implication rules that the *Read* authorization type for classes is implied by all other authorization types. Similar additional implication rules are derived for instances. They are listed below:

I_{I1} $\forall s: S, \forall I_i : Instance\ , \forall A_i : Att\text{-}set(I_i)\ \ (auth(s, I_i, Write) \Rightarrow auth(s, I_i, Write(A_i)))$

I_{I2} $\forall s: S, \forall I_i : Instance\ , \forall A_i : Att\text{-}set(I_i)\ \ (auth(s, I_i, Read) \Rightarrow auth(s, I_i, Read(A_i)))$

I_{I3} $\forall s: S, \forall I_i : Instance\ , \forall A_i : Att\text{-}set(I_i)\ \ (auth(s, I_i, Write(A_i)) \Rightarrow auth(s, I_i, Read(A_i)).$

Note that when a user receives the authorization to read an attribute of an object, and the value of this attribute is a reference to another object, this authorization does not imply in general that the user can read any attribute of the referenced object. He only reads the object-identifier of the referenced object.

Finally, note that we have also authorization types for databases. For completeness, we make the assumption that several databases may exist in the same system. In particular, the authorization type *Read* for databases allows users to read the database directory. That is, users can see the names of all classes within the database, and the inheritance hierarchies. However, this authorization type does not allow users to see the definition of specific classes. The reason for this is that in many applications, like CAD, often there is no much difference between classes and instances. Therefore, even a class definition may carry sensitive information which need to be protected. Note, instead, that the *Read-all* authorization type allows a user to read all objects within the database. This means that the user can read both class definitions and instances. We do not list here implication rules for database authorization types, since they are straightforward (and similar to the ones already defined for classes).

We remark now some differences with respect to the authorization model defined in [Rabi 91]. The model defined in [Rabi 91] seeks to minimize the number of authorization types. Therefore, only four authorization types were used in [Rabi 91]. This approach has two major disadvantages. The first is that a number of objects (called *authorization objects*) not corresponding to any real database object had to be introduced in the model. For example, in [Rabi 91] there is no authorization type *Read-all*. Therefore, in order to grant a user the right to read all instances of a given class, a special authorization object is introduced in [Rabi 91], called *setof-instances*, (there is one of such object for each class in the

schema). A *Read* authorization granted on this object simulates the authorization of reading all instances. In general this approach is quite unnatural, since it overloads the authorization types. As matter of fact, the end-user authorization language defined for the model of [Rabi 91] contains additional authorization types that are then mapped to the model by introducing those authorization objects. By contrast, in the approach proposed here we have extended the number of authorization types and reduced the number of object types, so that the objects used in authorizations correspond to real database objects. The major disadvantage of the approach proposed here is that we have a larger number of implication rules among authorization types. However, note that the number of implication rules is still quite small. For example, we have 8 implication rules for classes, while in [Rabi 91] this number is 4. However, the complexity due to these additional rules can be simply handled by implementing the authorization mechanism (or part of it) in Prolog (or other logical language).

Another difference is that in our model the authorization of modifying a class definition does not imply the authorization of modifying instances of the class, while in the model of [Rabi 91] it does so. The motivations for our approach are based on the fact that many class modification operations (such as adding or removing or changing methods) do not impact instances. Moreover, class modification operations that impact the instances (such as dropping or adding an attribute) do actually modify the instance structures, not their contents. Therefore, since the *Write* authorization type in our model means updating the contents of all attributes of an instance (not the instance structures), there should be no logical implication between the authorization to modify the definition of a class and to update its instances. Finally, another difference is that our model has provision for authorization types allowing reading/writing selected attributes for all instances of a class. This authorization type is not provided in the model defined in [Rabi 91]. It is, however, very useful; for example SQL/DS [SQL 81] allows update authorizations to be granted on selected columns of tables.

3.3 Authorization Objects

A third dimension along which authorizations can be propagated is along authorization objects. From authorization point of view, it is useful to see authorization objects organized into an *object authorization granularity hierarchy*. A node in the object authorization granularity hierarchy corresponds to a database object. Each node has associated authorization information about itself, and moreover it may have authorization information about a set of lower level nodes. For example, a class may have authorization information about reading or writing the class definition, but also authorization about reading or writing its instances. In some situations, it may be useful to factorize in the class an authorization holding on all instances of a class. The organization of objects into a granularity hierarchy allows authorizations associated with a node to be propagated to lower level nodes.

We start with a simple granularity hierarchy consisting of only three levels, corresponding respectively to databases (the topmost level), classes, and instances. Therefore, a database is seen as a set of classes, each class in turn is seen as a set of instances. This granularity hierarchy will be extended later when dealing with the additional modeling constructs such as versions. An example of granularity hierarchy is illustrated in Figure 3 for the classes defined in Figure 1. Note that with

respect to the object granularity hierarchy presented in [Rabi 91], the hierarchy presented here is simpler. This is due to the approach we took of increasing the set of authorization types. This resulted in a granularity hierarchy with less levels.

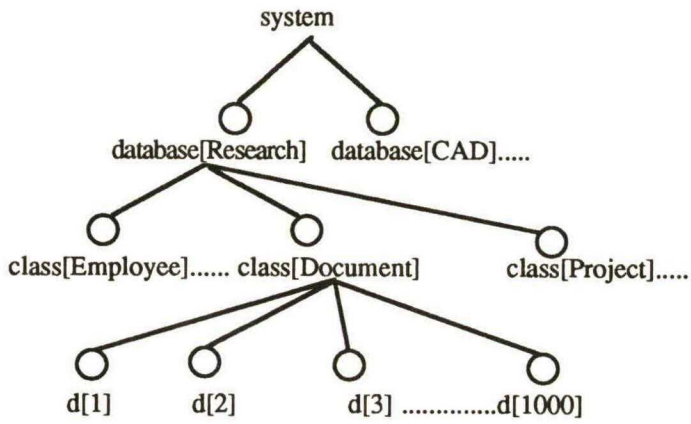

Figure 3. Example of object authorization granularity hierarchy

We now present the implication rules for objects. In the remainder of the discussion, given two object o_i and o_j, we use the notation $o_i \in o_j$, to indicate that o_i is an immediate descendant of o_j in the object granularity hierarchy. The notation means that o_i belongs to the set of objects represented by o_j. In presenting the implication rules, we first present those holding between the database level and the class level, and then between the class level and the instance level.

Implication rules between databases and classes

I_{DC1} $\forall s: S, \forall D_i : Database, \forall C_i : Class\ ((auth(s, D_i, Read\text{-}all) \wedge C_i \in D_i) \Rightarrow auth(s, C_i, Read\text{-}all))$

(Note that this rule together with rule I_{C1} for authorization types implies that a user holding the authorization to read all instances in the database may also read the definitions of all classes in the database.)

I_{DC2} $\forall s: S, \forall D_i : Database, \forall C_i : Class\ ((auth(s, D_i, Write\text{-}all) \wedge C_i \in D_i) \Rightarrow auth(s, C_i, Write\text{-}all))$

I_{DC3} $\forall s: S, \forall D_i : Database, \forall C_i : Class\ ((auth(s, D_i, Write\text{-}all) \wedge C_i \in D_i) \Rightarrow auth(s, C_i, Delete))$

I_{DC4} $\forall s: S, \forall D_i : Database, \forall C_i : Class\ ((auth(s, D_i, Write\text{-}all) \wedge C_i \in D_i) \Rightarrow auth(s, C_i, Write))$

I_{DC5} $\forall s: S, \forall D_i : Database, \forall C_i : Class\ ((auth(s, D_i, Write\text{-}all) \wedge C_i \in D_i) \Rightarrow auth(s, C_i, Create))$

Note from the above rules that the *Write-all* authorization type for databases is quite powerful, since it allows a user holding an authorization of this type to perform all possible actions on the database objects. Therefore, this authorization type is suitable mainly for users like database administrators. It can be seen as an authorization like to DBA authorization of SQL/DS [SQL 91].

Implication rules between classes and instances

I_{CI1} $\forall s: S, \forall C_i : Class, \forall I_i : Instance\ ((auth(s, C_i, Read\text{-}all) \wedge I_i \in C_i) \Rightarrow auth(s, I_i, Read))$

I_{CI2} $\forall s: S, \forall C_i : Class, \forall I_i : Instance\ ((auth(s, C_i, Write\text{-}all) \wedge I_i \in C_i) \Rightarrow auth(s, I_i, Write))$

I_{CI3} $\forall s: S, \forall C_i : Class, \forall I_i : Instance, \forall A_i : Att\text{-}set(C_i)$

$((auth(s, C_i, Read\text{-}all(A_i)) \wedge I_i \in C_i) \Rightarrow auth(s, I_i, Read(A_i)))$

I_{CI4} $\forall s: S, \forall C_i : Class, \forall I_i : Instance, \forall A_i : Att\text{-}set(C_i)$

$$((auth(s, C_i, Write\text{-}all(A_i)) \wedge I_i \in C_i) \Rightarrow auth(s, I_i, Write(A_i)))$$

Note that in rules I_{CI3} and I_{CI4} we do not need to state the constraint that attribute A_i be a valid attribute for instance I_i since I_i is an instance of class C_i and therefore all instance attributes defined by the class are valid attributes for the instances. However, some data models, like for example O_2 [Deux 90], allow instances to have additional attributes (and methods) with respect to those defined by their class. Authorizations on these additional attributes cannot be granted at class level (unless the authorization types *Read-all* or *Write-all* are used), since they are not common to all the instances of the class. Authorizations on them may only be granted at instance level. Implications rules holding for authorization types defined for instances (cf. Subsection 3.2) still apply to these exceptional attributes. Therefore, in the rules defined in Subsection 3.2, the attribute set of an instance I_i (denoted as $Att\text{-}set(I_i)$) includes the attributes defined by the class of I_i and additional attributes defined for I_i at instance level. Note that to deal with these additional attributes, we might have to introduce additional authorization types. For example, how is a user authorized to extend an instance by introducing additional attributes or methods? However, in order to keep the model simple, we make the assumption that the authorization type *Write* for instances also allow a user to add/remove additional attributes and methods.

All previous implication rules allow authorizations to be propagated top-down. We introduce now an implication rule that propagates bottom-up.

I_{CI5} $\forall s: S, \forall C_i : Class, \forall I_i : Instance, \forall A_i : Att\text{-}set(C_i)$ $((auth(s, I_i, Read(A_i)) \wedge I_i \in C_i) \Rightarrow auth(s, C_i, Read))$

This implication rule states that if a user has the *Read* authorization on an attribute of an instance, then he can read the class definition. The reason for this is that in order for a user to read an attribute of an instance, he must know the definition of the attribute. Therefore, he must be able to read the class definition, that contains the definition of the attribute. Note that we could further refine the model by introducing authorization types for reading/writing the definition of a single attribute. However, we do not think that such a fine level of granularity is really necessary.

Combining this rule with rules defined in Subsection 3.2 for authorization types of instances, we obtain that a user who can read an instance is also able to read the class definition. Note that the above rule I_{CI5} is quite useful, since it allows the authorization model to present a concise interface to the user. In this case, a user wishing to grant another user the authorization to read an object does not need to also grant the authorization to read the definition of the class, since this is automatically deduced by the system.

Content-dependent authorizations

In addition to the previous implication rules for content-independent authorizations, an object may factorize content-dependent authorizations for the set of its lower level objects in the granularity hierarchy. In other words, content-dependent authorizations can be expressed intensionally over a set of objects, rather than expressing them explicitly for each object. Content-dependent authorizations are expressed using the constraint language defined in Section 2.

In general, given an object O, these constraints contain a quantification of the form: $\forall O_i: O$. This quantification states that the conditions contained in the constraint apply to all objects belonging to O. Note that O can be a class, the most common case, or also a database (later on, we will see that this applies to versioned objects as well). When the content-

dependent authorization is associated with a database node, then it applies to all classes that are part of the database. In this case, the conditions must be expressed in terms of class-attributes that are common to all classes in the database. Class-attributes are those attributes that characterize the classes themselves as objects and are not inherited by the class instances. In our reference model, we do not include class-attributes. However, as an example, suppose that each class has a class-attribute called 'definition-date': this attribute contains the date when the class was defined. An example of content-dependent authorization associated with a database would be the following:

$\forall m$: *Manager*, $\forall C_i$: *Database[Research]* (C_i .*definition-date* > '*10-10-91*' $\Rightarrow$ *auth*(m, C_i, *Write*)).

This content-dependent authorization states that a manager can modify the class definition only for classes defined later than the specified date. However, more significant content-dependent authorizations are defined on classes.

Content-dependent authorization can be combined with implication rules for content-independent authorizations. Thus, implicit content-dependent authorizations are derived. As an example, consider the content-dependent authorization:

$\forall m$: *Manager*, $\forall d$: *Document* ($m \in d$.*authorlist* $\Rightarrow$ *auth*(m,d,*Write*)) (i).

By applying implication rules for authorization types holding for instances, we derive from (i) the following:

$\forall m$: *Manager*, $\forall d$: *Document* ($m \in d$.*authorlist* $\Rightarrow$ *auth*(m,d,*Read*)).

Moreover, by applying implication rule I_{CI5} holding between instances and their class, we obtain the following implicit content-based authorization:

$\forall m$: *Manager*, $\forall d$: *Document* ($m \in d$.*authorlist* $\Rightarrow$ *auth*(m, *Document*, *Read*)).

It is also possible to specify a content-dependent authorization for a specific instance. This is mainly useful when the instance changes content very often. Suppose that there is a project which is confidential and therefore the authorization policy is that authorizations for all documents related to that project must be explicitly stated using explicit authorizations (that is, authorizations must be decided case by case). However, suppose that a specific document, say d[3], is an overview of the project, reporting the main results every month, and must be made available for reading to all employees whenever the status is 'released'. A possible solution is to grant an explicit authorization to the role Employee each time d[3] is released and revoking this authorization each time the status is different from 'released'. This approach is quite cumbersome. A simpler solution is to define a content-dependent authorization of the form:

$\forall e$: *Employee* (*d[3].status='released'* $\Rightarrow$ *auth*(e, *d[3]*, *Read*)).

Therefore, content-dependent authorization provides a simple and effective solution to handle situations like the above one.

3.4 Additional Authorization Rules for Object-Oriented Data Models

We now describe how authorization is handled for additional modeling constructs. We will consider the cases of composite objects and versions. The authorization model should allow them to be handled as authorization units. For example, an authorization on the root of a version hierarchy should propagate to all versions. As we will see this is easily supported by the authorization model we have defined. Moreover, we briefly discuss authorization issues concerning user-defined methods.

Composite Objects

In general, when dealing with composite objects it is convenient handling them as an authorization unit. This allows a single authorization granted on the root of a composite object to be propagated to all components, without any additional explicit authorization. Before introducing the authorization approach used for composite objects, we need to introduce some additional notation. Given an instance I, *comp-set(I)* denotes the set of all instances that are components of I. Figure 4 presents some example of composite objects from the Document class of Figure 1. In Figure 4, there are two documents, identified respectively by d[1] and d[15]. They share a section identified by s[14].

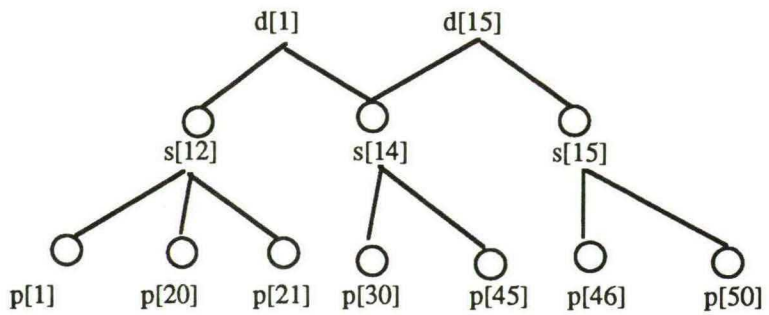

Figure 4. Composite object examples

In order to handle authorizations on composite objects, few authorization types and implication rules are added. They are as follows:

I_{Comp1} $\forall s: S, \forall I_i, \forall I_j : Instance$

$((auth(s, I_i, Read\text{-}composite) \wedge I_j \in comp\text{-}set(I_i)) \Rightarrow auth(s, I_j, Read\text{-}composite))$

I_{Comp2} $\forall s: S, \forall I_i, \forall I_j : Instance$

$((auth(s, I_i, Write\text{-}composite) \wedge I_j \in comp\text{-}set(I_i)) \Rightarrow auth(s, I_j, Write\text{-}composite))$

By applying those two rules recursively we have that authorization on the root of a composite object is propagated to all instances that are direct or indirect components of the root object. Note that the *Read* (*Write*) authorization type on an instance is different from the *Read-composite* (*Write-composite*). In the first case, a user can only read (write) all attributes of an object, including composite attributes. However, he cannot read (write) the content of referenced objects, even those that are referenced through composite attributes. In the second case, a user can read all attributes of an object and (recursively) all attributes of objects that are referenced through composite references. Therefore, the *Read-composite* (*Write-composite*) authorization type is stronger than the *Read* (*Write*) authorization type. This is formalized by the following implication rules among authorization types for instances:

I_{I6} $\forall s: S, \forall I_i : Instance$ $(auth(s, I_i, Write\text{-}composite) \Rightarrow auth(s, I_i, Write))$

I_{I7} $\forall s: S, \forall I_i : Instance$ $(auth(s, I_i, Read\text{-}composite) \Rightarrow auth(s, I_i, Read))$.

Similarly to the case of authorization types *Read* and *Write*, we establish an implication rule stating that a user who can modify a composite object is also able to read it:

I_{I8} $\forall s: S, \forall I_i : Instance$ $(auth(s, I_i, Write\text{-}composite) \Rightarrow auth(s, I_i, Read\text{-}composite))$.

Combining the rules for composite objects with the rules defined for classes and instances, we obtain that an authorization to read (write) a composite object implies the authorization to read the class definitions of all the component instances. However, no implicit authorizations are derived on instances of component classes that are not used as components of the composite object. As an example consider the composite objects of Figure 4. Suppose that the following explicit authorization is entered: *(s_i, d[1], Read-composite)*. From this authorization, the following implicit *Read* authorizations are derived:

(s_i, s[12], Read), (s_i, s[14], Read), (s_i, p[1], Read), (s_i, p[20], Read),

(s_i, p[21], Read), (s_i, p[30], Read), (s_i, p[45], Read).

Moreover, the following authorization on classes Section and Paragraph are derived:

(s_i, Section, Read), (s_i, Paragraph, Read).

However, no authorizations are implied on instance s[15] of class Section and on instances p[46] and p[50] of class Paragraph. The above approach for composite objects is in some way similar to the approach used in [Garv 91], where it is stated that the access class of an object dominates that of its components.

In addition to grant authorizations on a single composite object, it may useful to grant authorizations for composite objects at class level. Therefore, we introduce also for classes two new authorization types, *Read-composite-all* and *Write-composite-all*. Their meaning is explained by the two following implication rules that we add to the set of implication rules between classes and instances stated in Subsection 3.3.

I_{CI6} $\forall s: S, \forall C_i : Class, \forall I_i : Instance ((auth(s, C_i, Read\text{-}composite\text{-}all) \wedge I_i \in C_i) \Rightarrow auth(s, I_i, Read\text{-}composite))$

I_{CI7} $\forall s: S, \forall C_i : Class, \forall I_i : Instance ((auth(s, C_i, Write\text{-}composite\text{-}all) \wedge I_i \in C_i) \Rightarrow auth(s, I_i, Write\text{-}composite))$.

An additional implication rule holding between *Read-composite-all* and *Write-composite-all* states that the second authorization type implies the first. We do not formalize it, since it is quite similar to the one stated for authorization types *Read-all* and *Write-all*.

Since a composite object represents also a set of components in the object granularity hierarchy, it would be possible in principle to associate with the root of the composite objects content-dependent authorizations for the components, as we do for classes and instances. Here, however, the situation is different since component objects may be of different classes, and therefore their types may be different. Therefore, it is difficult to devise situations when this may arise in practice. As an example, suppose that a date is associated with each component of documents. Then a possible content-dependent authorization would be that an employee can read all components of document d[20] if their date is lower than '27-March-91'. This authorization could be modeled as follows:

$\forall e: Employee, \forall I_i: comp\text{-}set(d[20])\ (I_i.date < '27\text{-}03\text{-}91' \Rightarrow auth(e, I_i, Read\text{-}composite))$.

Consider now the following authorization policy:

An employee can read a section of document d[20] only if the date of the section is earlier than '27-March-91'.

The difference with the previous authorization is that here only the components that are sections are considered (that is, the authorization does not apply to the abstract). This authorization is expressed as follows:

$\forall e: Employee, \forall s: Section\ ((s \in comp\text{-}set(d[20]) \wedge s.date < '27\text{-}03\text{-}91') \Rightarrow auth(e, s, Read\text{-}composite))$.

Note that in the above example, the authorization is really associated with the Section class. The constraint that this authorization only holds for components of document d[20] is simply expressed as a set-membership predicate. If however this policy were to be applied to each Section, independently on the document where it is contained, then it would simply be expressed as:

$\forall e$: *Employee*, $\forall s$: *Section* (*s.date* < '27-March-91' $\Rightarrow$ *auth(e,s,Read-composite)*) .

Finally, we present another example showing an authorization based on the content of both the root object and on the components. Consider the following authorization policy:

An employee can read a section of a document only if the employee is in the authorlist of the section and the document is related to an ESPRIT project.

This policy would be formulated as follows:

$\forall e$: *Employee*, $\forall s$: *Section*, $\forall d$: *Document* $((s \in comp\text{-}set(d) \wedge$

$e \in s.section\_authors \wedge d.project.research\_programme$ = 'ESPRIT') $\Rightarrow$ *auth(e, s, Read-composite)).*

The above examples show the flexibility of the constraint language for handling a large variety of applicative situations.

Versions

A versioned object may potentially have a large number of versions. Therefore, as for composite objects, it should be possible to handle versioned objects as authorization units. For example, it should be possible to grant a single read on a versioned object allowing users to read all versions of this object without any additional explicit authorization. As for composite objects, given a generic instance *I version-set(I)* denotes the set of all instances that are versions of *I*. Therefore, *version-set(I)* includes all objects that belong to the version hierarchy rooted at *I*. To handle authorizations on versioned objects additional implication rules are introduced. Note that those implication rules are simply formulated using a membership predicate to test whether an instance belongs to a version hierarchy of a generic instance. The implication rules for versions are as follows:

I_{Vers1} $\forall s: S, \forall I_i, I_j : Instance$ $((auth(s, I_i, Read) \wedge I_j \in version\text{-}set(I_i)) \Rightarrow auth(s, I_j, Read))$

I_{Vers2} $\forall s: S, \forall I_i, I_j : Instance$ $((auth(s, I_i, Write) \wedge I_j \in version\text{-}set(I_i)) \Rightarrow auth(s, I_j, Write)).$

Similar implication rules are derived for authorization to read or write specific attributes in all versions of a given version hierarchy:

I_{Vers3} $\forall s: S, \forall I_i, I_j : Instance, \forall A_i : Att\text{-}set(I_i)$

$((auth(s, I_i, Read(A_i)) \wedge I_j \in version\text{-}set(I_i)) \Rightarrow auth(s, I_j, Read(A_i)))$

I_{Vers4} $\forall s: S, \forall I_i, I_j : Instance, \forall A_i : Att\text{-}set(I_i)$

$((auth(s, I_i, Write(A_i)) \wedge I_j \in version\text{-}set(I_i)) \Rightarrow auth(s, I_j, Write(A_i))).$

An important question for versioned objects concerns authorization for modifying version hierarchies. Several operations, such as promoting a transient version or changing the default, modify the version hierarchy. One possibility is to introduce new authorization types, specific for those operations. However, this solution seems unnecessarily complex. A

simpler solution is to use the *Write* authorization on the generic instance to this purpose as well. This approach is very simple and it is consistent with the fact that the generic instance in a version hierarchy stores all information about the version hierarchy. Therefore, a *Write* authorization on the generic instance of a version object allows users to perform all "administration" functions concerning the version hierarchy.

Another question related to version hierarchies is the authorization to derive a new version within a given version hierarchy. A possible solution is to use the same approach we used for operations concerning the version hierarchy administration. Under this approach a user would have to receive the *Write* authorization on the generic object in order to be able to create a new version. This approach has the main disadvantage that then that user would be able to write all versions in the version hierarchy (remember from implication rule I_{Vers2} a *Write* authorization on the generic instance is implicitly propagated to all versions). However, a main reason for giving a user the possibility of creating a new version is to prevent the user from directly modifying some existing objects. Another possibility is to require that a user has the *Create* authorization type on the class in order to derive a new version. Again this solution is not completely satisfactory because this would allow a user to create new version hierarchies. In applications where there are a lot of objects, each with many versions, it may be important to control the proliferation of version hierarchies.

The approach we take consists of introducing a *Create* authorization type for instances as well. A *Create* authorization type must be associated with instances that belong to version hierarchies. The semantics of this authorization type for an instance *I* is to allow a user to create a new version within the version hierarchy rooted at *I*. Note that *I* can be a generic instance (that is, the root) of a version hierarchy or can be a specific version. The fact that *I* can also be a specific version is quite important since it allows the user to create versions only from *I*, or from versions derived from *I*. In this way it is possible to limit the user action to only a specific subhierarchy within an entire version hierarchy. Note that versions can only be derived from stable versions. Therefore a restriction is that *I* must be a stable version. Moreover, the user can create versions only from the stable versions that belong to the version subhierarchy rooted at *I*. We note that our approach is quite different in this respect from [Rabi 91], where the approach is taken that user must possess the *Write* authorization type on the generic instance in order to create new instances.

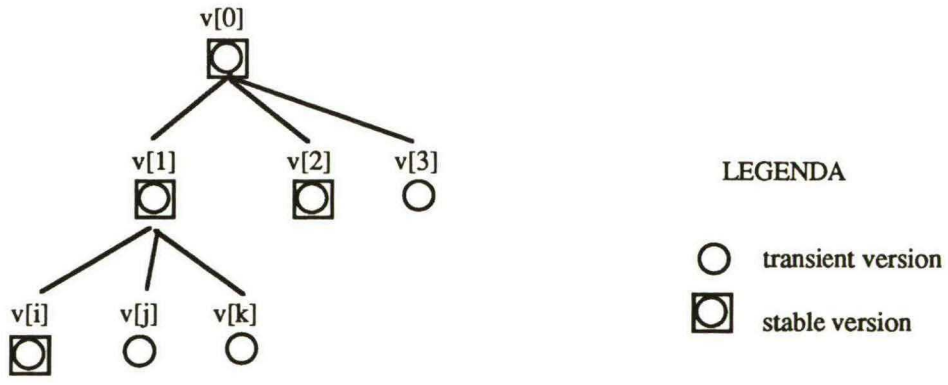

Figure 5. Version hierarchy example

As an example consider the version hierarchy in Figure 5 and the authorizations

auth(s_i, v[0], Create) auth(s_j, v[i], Create).

Under the previous authorizations, the first subject, s_i, is a authorized to create new versions from the versions v[0], v[1], v[2], v[i]; the second subject, s_j, can only derive versions from v[i]. Finally, we note that a user receiving the authorization to create a new version from a given object, or from versions of this object, must be able to read the object from which the version is to be derived. Therefore, an additional implicaton rule for versioned objects is added:

I_{Vers5} $\forall s: S, \forall I_i : Instance \ ((auth(s, I_i, Create) \Rightarrow auth(s, I_i, Read))$.

Since a versioned object represents a set of instances, it is possible to also specify content-dependent authorizations that are applied to all versions in a given version hierarchy. In addition, the special predicate *Is_Stable* is introduced to test whether a version is stable or still transient. For example, suppose that the we wish to enforce that only stable versions of document d[3] can be read by the role Chiefaccountant (cf. Figure 2). This authorization can be simply specified as

$\forall c: Chiefaccountant, \forall d: Document \ ((d \in version\text{-}set(d[3]) \wedge Is\_Stable(d)) \Rightarrow auth(c,d,Read))$.

Inheritance hierarchies

Inheritance is a basic concept in many advanced data models. It provides several important functions, such reusability and conceptual modeling [Bert 91]. An important question concerning inheritance hierarchies is whether authorizations on a class should be inherited by its subclasses. An approach in which authorizations are automatically propagated to the subclasses has the advantage that a single authorization on the root of an inheritance hierarchy is sufficient for all classes in this hierarchy. Moreover, this approach is more efficient for queries applying to the members of a class, and not only to its instances (cf. Section 2.1 for the difference between an instance and a member of a class). However, as pointed out in [Rabi 91], this approach has a major disadvantage in that it discourages the usage of inheritance for re-usability. A user wishing to create a class re-using another class definition would hesitate in doing so, since all authorizations holding on the superclass would automatically hold on his/her class. Moreover, this approach does not allow to increase the protection of subclasses with respect to superclasses. Therefore, the approach adopted in [Rabi 91] and in here is that authorizations are not automatically inherited along class inheritance hierarchy. However, in some situations it may be useful to allow inheritance of authorizations. To this purpose, we introduce the relationship of authorization inheritance among classes. This relationship is denoted as $\preceq$; given two classes C and C' such that C is a subclass of C', $C \preceq C'$ indicates that all authorizations holding on C' automatically hold on C. This relationship is entered by the user, and by default does not hold among classes (we present user interface statements for this in Appendix A). We refine this relationship by distinguishing among inheritance of content-independent authorizations (denoted as $\preceq^{CI}$) and inheritance of content-dependent authorizations (denoted as $\preceq^{CD}$). The reason for distinguishing them is that content-dependent authorizations may not be valid on a subclass, because of overriding and multiple inheritance. As discussed in [Bert 91], depending on the specific object-oriented data model overriding and multiple inheritance may cause an attribute of a subclass to have a definition which is different from the attribute with the same name of the superclass. In this situation, content-dependent authorizations involving this attribute may not be valid any more because of type mismatch. This means that they have to be redefined in the subclass. In such cases, it may be useful to only inherit content-independent authorizations. When the relationship $\preceq$ holds between two classes, both content-dependent and content-independent authorizations are inherited.

To model the inheritance of authorizations, we introduce two additional implication rules:

I_{Inher1} $\forall s: S, \forall C, C': Class, \forall a: A\ ((b\_auth(s,C',a) \wedge C \leq^{CI} C') \Rightarrow auth(s,C,a))$

I_{Inher2} $\forall s: S, \forall C, C': Class, \forall a: A\ ((X(C') \wedge C \leq^{CD} C') \models X(C))$

Note that first implication rule, combined with the implication rules introduced earlier, allows several implicit authorizations to be derived. In the second implication $X(C')\ (\ X(C))$ denotes a content-dependent authorization expressed in terms of class C' (C). We now present some examples with respect to the inheritance hierarchy of Figure 6.

Figure 6. Inheritance hierarchy example

Suppose that the authorization inheritance relationship $Memo \leq Document$ holds. Moreover, suppose that the following authorizations hold on class Document:

b_auth(Manager, Document, Read-all))

$\forall e: Employee, \forall d: Document\ (e \in d.authorlist \Rightarrow auth(e,d,Read))$.

Because of the implications for inheritance hierarchies, we obtain that the following authorizations hold on class Memo:

auth(Manager, Memo, Read-all)

$\forall e: Employee, \forall d: Memo\ (e \in d.authorlist \Rightarrow auth(e,d,Read))$.

However, no similar implications are deduced for the subclasses ResearchReport and Paper since no authorization inheritance relationship has been entered for them.

A final question concerns authorizations to create a subclass of one or more classes. The approach taken in [Rabi 91] is to introduce a special authorization type. A user who has been granted this authorization type on a class C is able to create subclasses of it. We take a different approach in that no new authorization types are introduced. Rather, the already existing authorization types are used. A user can create a subclass from classes $C_1, C_2, \ldots, C_i$ $(i \geq 1)$ if he has the authorization to create a class and moreover he has the authorization to read the definitions of these classes. Note that in order to create a subclass, two different types of authorizations are needed. The first one is an authorization to occupy storage space and to add a new class to the database. This authorization is usually managed by some administrators (In SQL/DS for example, a user can create a table only if he has the resource authority which can be granted only by DBAs.) The second type of authorization is the authorization to read the definitions of superclasses. This authorizations are usually managed by the creators of classes, who are not however concerned in most cases with storage space administration.

Authorization for user-defined methods

In the previous discussion we have only considered authorizations for system defined methods. An important question is how authorization is dealt with for user-defined methods. A discussion of issues concerning this question can be found in [Bert 92]. Basically, in [Bert 92] two possible approaches are devised. Under the first approach, when a user-defined method is executed, all accesses performed by this method are checked for authorization against the user on behalf of whom the

method is being executed. In other words, in order to execute a method a user must possess all authorizations needed by the method during its execution. If we consider the class Document of Figure 1 and the method Copy, under this approach a user will need the authorization to read the document which is being copied and the authorization to create a new instance of the class Document. This approach is used in the model defined in [Rabi 91]. The main problem of this approach is that it does not exploit the potential of methods for authorization. In certain situations, it may be useful that a user be able to execute a method, without having direct access to the object the method accesses. This approach is called in [Bert 92] *execution under protection mode*. All authorizations needed during method executions are checked against a user different from the user *u* who started the method execution. The user against whom authorizations are checked is the user who granted *u* the authorization to execute the method. Usually, that user is one of the owners of the object on which the method has been invoked. This protection mechanism is quite important and it is used, under different forms, in relational DBMSs, file systems, and operating systems. Note that the authorization mechanisms should support both ways of method execution (non-protection and protection mode).

An important advantage of using methods for authorizations is that it is possible to embed into methods authorization checkings. Since methods are expressed in computationally complete languages it is possible to encode whatever complex authorization rules. This feature coupled with the possibility of running methods under protection mode allows objects to be encapsulated with methods enforcing authorizations.

An important issue, that we leave open, is whether the content-dependent authorization mechanism defined here is redundant when user-defined methods implement authorization rules as part of their execution. Note that it is possible to implement content-dependent authorization using methods. A main difference between the two is that content-dependent authorization rules defined by the constraint language are declarative, while authorization rules defined as part of methods are expressed in an imperative language. The usage of the constraint language simplifies the definition of authorization rules by users, and saves the users from writing several methods. However, the expressive power of the constraint language is limited with respect to the power of a general programming language. Therefore, both declarative content-dependent and procedural content-dependent authorization seem to be useful. However, more investigations are needed on this question. Related to this there is the definition of methodological guidances supporting the authorization administrators and database designers in the task of designing authorization rules for a given database.

4. Authorization Administration

To be complete an authorization model must include policies and mechanisms for adminstration of authorizations. An administration policy states which users are able to perform authorization operations, such as grant and revoke. Different policies are implemented by different systems. An approach is to rely on the database administrator (DBA) for granting and revoking authorizations (we call this approach *centralized administration*). Under this approach, a user who has created an object is not able to autonomously managing authorizations on his own objects. This problem is solved by the *ownership approach*. Under this approach, the owner of an object is the only user able to grant/revoke authorizations on his own objects. The owner of an object is usually the user who has created the object. The ownership approach is very often

extended with the possibility of delegating the administration of authorizations, that is, the owner of an object may allow other users to grant/revoke authorizations on this object. This approach has been used by various relational systems, such as System R [Grif 76], and has many advantages.

In the ownership approach the unit of ownership is the relation. In object-oriented databases, however, the unit of ownership cannot be the entire extension of a class. Rather, it should be possible that different instances of the same class have different owners. This requirement is motivated by the fact that the class often represents just a template from which several users derive their own instances. In this case, it may quite restrictive to impose a single user as the owner of all instances of a class. Therefore, in object-oriented databases there is the need of further decentralization allowing class instances be independently administrated. If the authorization model were not to allow single instances to be independently administrated, in some situations users would have to create subclasses only for the purpose of being able to administrate their own instances. The model presented in the previous sections is then extended by associating an owner to each object. Then it is possible to apply mechanisms, like those defined in [Grif 76], for the delegation of administration functions. For example, the owner of an object may grant another user an authorization with grant option, thereby allowing this user to grant this authorization to other users. A main difference, however, between the model of [Grif 76] and the model presented here and in [Bert 92], is that we allow the owner of an object to change[4]. This is very useful in applications dealing with cooperative work and long-lived objects. As in [Grif 76], the owner of an object is the user who initially created the object. However, in our model this owner may change later on.

We note, however, that the decentralization at instance level may sometimes be inconvenient for perfomance and also conciseness. We discussed earlier that different granularity levels should be supported by the authorization mechanism. We have shown that it is possible to grant the same authorization on all instances of a class; examples are the *Read-all* and *Write-all* authorization types. However, when each instance of a class is independently administrated, it is not clear which user can grant such an authorization. Also, the capability of inheriting authorizations from superclasses may cause problems since it is not clear which user can issue such command. However, authorization types holding on all instances of a class are very important for performance, since in the case of queries a single authorization check is sufficient, and for conciseness, since a single authorization command from the user is needed. To solve this problem, we introduce the possibility for classes of being centrally administrated. This approach consists of having a single owner of all instances of a class (called *class administrator*). Therefore, instances of a class may be created by different users, but authorizations are centrally managed by the class administrator. In our model, the administration modality (*decentralized* vs *centralized*) for a class is declared when the class is defined. It is, however, possible to change the administration modalities later on. In particular, when the class is centrally administrated, the class administrator can change the administration from centralized

[4] Note that one reason why in the authorization model defined in [Grif 76], the owner of a table cannot change is because full names for tables include the owner name. This problem does not hold in OODBMSs where objects have unique OIDs, which do not include the name of the owner.

to decentralized. Changing the class administration from decentralized to centralized requires the authority of a database administrator (DBA) that can name the proper class administrator[5].

5. Conclusions

In this paper we have presented an authorization model for object-oriented databases. A major difference of our model, compared to other models like the one presented in [Rabi 91], is that our model provides also content-dependent authorization. The possibility of establishing authorization rules based on object contents has several advantages. First, it is easier for the users to formulate authorization rules, since these can be directly derived from the overall organization authorization policies. Therefore, the authorization rules are strictly related to the application semantics. Moreover, they do not require the users to grant and/or revoke authorizations when objects change their status, since authorizations hold as long as objects verify authorization predicates. If an object has changed its status and verifies an authorization predicate, the system automatically allows authorized users to access the object. Similarly, if an object does not verify an authorization predicate, accesses to this object under this authorization predicate are not any longer possible. In both situations, there is no need of explicit grant or revoke operations. Finally, content-dependent authorization is more efficient with respect to query execution, since authorization predicates can be merged with predicates in queries, as done for views in relational systems. Within the paper we have shown several examples of the flexibility provided by content-dependent authorization for semantic modeling concepts such as versions and composite objects.

In addition, the model presented in this paper differs from the model defined in [Rabi 91] in a number of other aspects. First, we took the approach of increasing the authorization types of the model and reducing the number of authorization objects, while in [Rabi 91] only four authorization types are used while the number of authorization objects is quite high. The main drawback of the model in [Rabi 91] is that authorization objects are introduced that do not always correspond to real database objects. The model presented here is therefore more natural and can be easily mapped onto a user language. Moreover, our model allows a more detailed modeling of implications among authorization types. Second, we have introduced a more sophisticated authorization model for versions and for inheritance hierarchies. Third, we have presented two possible complementary approaches to authorization administration. It is our opinion that both are needed to provide a high degree of flexibility.

References

[Andr 87] Andrews, T., Harris, C. "Combining language and database advances in an object-oriented development environment" *Proc. Second International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, Oct. 1987.

---

[5] Another possibility to be explored is to have both a class administrator able to grant authorization types such as *Read-all* and *Write-all*, and able to issue authorization inheritance commands, and different owners for instances; therefore, combining the two approaches.

[Atki 89] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich., K., Maier, D., Zdonik, S. "The object-oriented database system manifesto" *Proc. First International Conference on Deductive and Object-Oriented Databases*, Kyoto (Japan), Dec. 4-6, 1989.

[Bert 88] Bertino, E., Haas, L.M. "Views and security in distributed database management systems" *Advances in Database Technology, Proc. First International Conference on Extending Database Technology (EDBT)*, Venice (Italy), March 14-18, 1988, Lecture Notes in Computer Science 303, Springer-Verlag.

[Bert 91] Bertino, E., Martino, L. "Object-oriented database management systems: concepts and issues" *Computer* (IEEE Computer Society), Vol.24, No.4 (1991), 33-47.

[Bert 92] Bertino, E. "Data hiding and security in object-oriented databases" accepted to the *Eighth IEEE International Conference on Data Engineering*, Phoenix (Ariz.), Feb. 3-7, 1992.

[Bret 89] Breitl, R., et Al. "The GemStone data management system" *Object-Oriented Concepts, Databases, and Applications*, W. Kim, and F. Lochovsky, eds., Addison-Wesley (1989), 283-308.

[Deux 90] Deux, O. et Al. "The story of O_2" *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, N. 1 (1990), 91-108.

[Fish 87] Fishman, D.H., et Al. "IRIS: an object-oriented database management system" *ACM Trans. on Office Information Systems*, Vol. 5, N. 1 (1987).

[Fern 81] Fernandez, E., Summers, R.C., Wood, C. *Database security and integrity*. Addison-Wesley, Reading (Mass.), 1981.

[Garv 91] Garvey, T.D., and Lunt, T.F. "Multilevel Security for Knowledge Based Systems" Technical Report SRI-CSL-91-01, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[Gass 88] Gasser, M. *Building a secure computer*. Van Nstrand Reinhold, New York, 1988.

[Grif 76] Griffiths, P.P., and Wade, B.W. "An authorization mechanism for a relational database system" *ACM Trans. on Database Systems*, Vol.1, No.3 (1976), 242-255.

[Jajo 91] Jajodia, S., and Sandhu, R. "Toward a multilevel secure relational model", *Proc. ACM-SIGMOD International Conference on the Management of Data*, Denver (Color.), May 29-31,1991.

[Katz 90] Katz, R. "Toward a unified framework for version modeling in engineering databases", *ACM Computing Surveys*, Vol.22, No.4 (1990), 375-408.

[Keef 89] Keefe, T.F., Tsai, W.T., and Thuraisingham, M.B. "SODA: a secure object-oriented database system" *Computer & Security*, Vol.8 (1991), 517-533.

[Kim 89] Kim, W., Bertino, E., Garza, J.F. "Composite objects revisited" *Proceedings ACM-SIGMOD International Conference on the Management of Data*, Portland (Or.), May-June 1989.

[Kim 90] Kim, W. "Object-oriented databases: definitions and research directions" *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, N.3, (1990), 327-341.

[Rabi 91] Rabitti, F., Bertino, E., Kim, W., Woelk, D. "A model of authorization for next-generation database systems" *ACM Trans. on Database Systems*, Vol. 16, No.1 (1991), 88-131.

[SQL 81] IBM Corporation. SQL/Data System: Application Programming, SH24-5018 (1981).

[Weig 91] Weigand, H. "An object-oriented approach in a multimedia database project" Kent, W., Meersman, R. (eds.), *Object-oriented databases* (DS-4), W. Kent, and R. Meersman, eds., North-Holland, Amsterdam, 1991.

[Wier 91] Wieringa, R.J., Weigand, H., Meyer, J.-J.Ch, Dignum, F.P.M. "The inheritance of dynamic and deontic integrity constraints" *Annals of Mathematics and Artificial Intelligence*, Vol. 3 (1991), 393-428.

Appendix A - Authorization language

Here we sketch an SQL-like language for authorization operations grant and revoke. The basic format of the grant command is:

GRANT <authorization> ON <object> [WHERE <qualification-clause>] TO <subject>

In the previous command, the possible authorization types are:

READ, READ-ALL, WRITE, WRITE-ALL, DELETE, CREATE, READ-COMPOSITE, WRITE-COMPOSITE, READ-COMPOSITE-ALL, WRITE-COMPOSITE-ALL

<object> can be a database name, a class name, an instance name, a class name followed by a set of column names, an instance name followed by a set of column name;

<qualification-clause> is a SQL-like formulation for the constraint language presented in the paper; it is used to express content-dependent authorizations;

<subject> can be role or a single user.

REVOKE <authorization> ON <object> FROM <subject>

In addition to this basic format, we have some special formats for the grant and revoke commands that have been introduced to deal with authorization aspects that are peculiar to our model. They are as follows:

GRANT ALL | BASE | CONTENT ON Class_name_1 AS Class_name_2

This format of grant command allows the inheritance of authorizations from the class with name Class_name_2 to the class with name Class_name_1. It is possible to inherit all authorizations (option ALL), or only the content-independent authorizations (option BASE), or only the content-dependent authorizations (option CONTENT). It is possible to revoke the inheritance by using the following format of the revoke command:

REVOKE ALL | BASE | CONTENT ON Class_name_1 FROM Class_name_2

This command has the effect of suppressing the inheritance of authorizations from the class with name Class_name_2 to the class with name Class_name_1. It is possible to selectively suppress the inheritance of only the content-independent authorizations, or of only the content-dependent authorizations.

Other commands that we do not present here include commands for ownership and class administration, and for role management.

OVERVIEW OF ITK RESEARCH REPORTS

No	Author	Title
1	H.C. Bunt	On-line Interpretation in Speech Understanding and Dialogue Sytems
2	P.A. Flach	Concept Learning from Examples Theoretical Foundations
3	O. De Troyer	RIDL*: A Tool for the Computer-Assisted Engineering of Large Databases in the Presence of Integrity Constraints
4	E. Thijsse	Something you might want to know about "wanting to know"
5	H.C. Bunt	A Model-theoretic Approach to Multi-Database Knowledge Representation
6	E.J. v.d. Linden	Lambek theorem proving and feature unification
7	H.C. Bunt	DPSG and its use in sentence generation from meaning representations
8	R. Berndsen en H. Daniels	Qualitative Economics in Prolog
9	P.A. Flach	A simple concept learner and its implementation
10	P.A. Flach	Second-order inductive learning
11	E. Thijsse	Partical logic and modal logic: a systematic survey
12	F. Dols	The Representation of Definite Description
13	R.J. Beun	The recognition of Declarative Questions in Information Dialogues
14	H.C. Bunt	Language Understanding by Computer: Developments on the Theoretical Side
15	H.C. Bunt	DIT Dynamic Interpretation in Text and dialogue
16	R. Ahn en H.P. Kolb	Discourse Representation meets Constructive Mathematics

No	Author	Title
17	G. Minnen en E.J. v.d. Linden	Algorithmen for generation in lambek theorem proving
18	H.C. Bunt	DPSG and its use in parsing
19	H.P. Kolb	Levels and Empty? Categories in a Principles and Parameters Approach to Parsing
20	H.C. Bunt	Modular Incremental Modelling Belief and Intention
21	F. Dols en H. Daniels	Nog niet verschenen
22	F. Dols	Nog niet verschenen
23	P.A. Flach	Inductive characterisation of database relations
24	E. Thijsse H. Daniels	Definability in partial logic: the propositional part
25	H. Weigand	Modelling Documents
26	O. De Troyer	Object Oriented methods in data engineering
27	O. De Troyer	The O-O Binary Relationship Model
28	E. Thijsse	On total awareness logics
29	E. Aarts	Recognition for Acyclic Context Sensitive Grammars is NP-complete
30	P.A. Flach	The role of explanations in inductive learning
31	W. Daelemans, K. De Smedt en J. de Graaf	Default inheritance in an object-oriented representation of linguistic categories
32	E. Bertino H. Weigand	An Approach to Authorization Modeling in Object-Oriented Database Systems

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS