A Model of Content-based Authorization in Object-Oriented Databases based on Object Views

Ahmad Baraani-Dastjerdi *Josef Pieprzyk*
Reihaneh Safavi-Naini *Janusz R. Getta*

Department of Computer Science
University of Wollongong
Wollongong, NSW 2522
AUSTRALIA
e-mail: [ahmadb,josef,rei,jrg]@cs.uow.edu.au

July 26, 1995

Abstract

Several models of authorization have been proposed for object-oriented databases supporting different levels of granularity. However, these models do not support authorization based on database contents and context. A way of handling context and content-dependent authorization is by using views. In this paper, we present a model of authorization, based on a view model proposed by Bertino [4], that supports content-based access control on instances of a class.

Keywords: View mechanism, View hierarchy, Access view, Discretionary security, Implication rules, authorization system

1 Introduction

A *view* or *virtual class* can be used for protection by allowing subsets of data to be seen/modified by users with the required privilege. Views can be used to provide the desired level of granularity if a powerful enough query language is used for their definition. They also provide an object content-dependent authorization.

So far, several authorization models for object-oriented databases, which support discretionary or mandatory access control, have been defined [6, 8, 1, 2, 7, 9, 10, 11, 12, 17, 18, 19]. However, none of them support content-based access control on instances of a class. Recently, several authorization models based on methods, which support content-based authorization, have been proposed [13, 5]. This approach has the drawback that method specification would be dependent on authorizations, and therefore a change of the authorizations would require a change in the specification of the methods. Bertino and Weigand [3] have used constraint language to provide content-dependent authorizations for the authorization model proposed in [6]. A main problem of the model is how to evaluate conditions associated with authorizations efficiently in the model's implementation. In particular, enforcing satisfaction of the conditions expressed in the authorizations by filtering the data prior to user access would require a double access to the object (one to evaluate the conditions and the other to satisfy the user query). A way to provide this kind of authorization is to define views containing conditions on the values of specific instance-variables of the classes. This approach considered in relational database systems is simply to add the conditions expressed in the authorization to the user query, known as the query modification mechanism. This has the advantage of ensuring the satisfaction of the protection requirements and not overloading the access control. Moreover because definitions

of views can be used to restrict the instance-variables, methods, or object that are accessible to users, different levels of granularity such class-based, object-based, method-based, and instance-variable based authorizations can be provided.

In [4], Bertino presents a view model for object-oriented databases that extends in various directions typical view models of relational databases. Since, at present, there is not no consensus about the form of the view model for object-oriented databases, we use Bertino's view model and present a view-based authorization for object-oriented databases.

The primary purpose of this paper is to discuss the use of views in object-oriented database systems that enforces discretionary access control policies, and in particular addresses the problems of context and content-dependent authorizations. We first discuss in the context of the view model proposed in [4], the usage of views for user shorthand and protection. To improve the flexibility of the view model and consequently the operational security of the system, *parameterized views* are introduced that extends the view model in [4], with parameters which are bounded to the actual values at the time when a view is evaluated. Four types of inheritance relationship between base classes (or views) and derived views are defined: *specialization inheritance, constraint inheritance, strict constraint inheritance,* and *proper specialization inheritance.*

The paper is organized as follows. Section 2 provides basic concepts and definitions. Section 3 discusses how view hierarchies can be inferred from class hierarchies and view definitions. In Section 4, the *access view* for providing access control on views is introduced. Also security requirements for discretionary access control are given. Section 5 discusses implicit authorization rules for each domain of authorization (user, view, access right). Section 6 presents the way validity of an access request, according to the authorization system, can be checked. Finally, Section 7 summarizes the paper.

2 Basic Concept

In this section, we briefly review the view model proposed by Bertino [4] and present examples to illustrate some issues such as user shorthand, data protection, and in particular, view's effect on hierarchical structure.

2.1 Views

In the relational data model, a view is defined as a *virtual relation* derived by a query on one or more stored relations. The relational operations *join, select,* and *project* may be used to define a view. Views can be used in (almost) any query where relations can be used. Furthermore, authorizations may be granted and revoked on views as on ordinary stored relations. This possibility allows content-based authorization on stored relations. Therefore, views can be used for two purposes: *data protection* and *user convenience.*

Views in an object-oriented database model have been considered in a number of papers [4, 15, 16, 20]. However, at present, there is not no consensus about the form of the view model for object-oriented databases. Until such a standard becomes available, the choice of a view model is an arbitrary. We here consider the view model which was developed by Bertino [4] and describe how security properties might be incorporated into such model. It is worth noting that while the authorization model is developed in context of the view model proposed in [4], the essential ideas are widely applicable.

In [4], Bertino provides a view model for object-oriented databases that extends in various directions typical view models of relational databases. In this model a view is defined by a query, as in relational systems, on one or more classes, called *base class(es).* However, to make the view model suitable for usage in object-oriented databases several additional features are also provided. First, views with additional properties that are not derived from the base classes are introduced (**additionalproperties**). The property specifications in a view is similar to property specifications in a class definition; they consist of property names and domains; the domain can

be a class or a view. Second, the view model is not restricted to a structural model but it also has behavioral features, i.e., a view may have methods (**methods**). A method can be derived from the base class(es), with the same name or renamed, or can be a new method. Finally, as noted a view may contain additional properties and methods other than those derived from the base class(es). Hence in order to re-use existing view definitions in the specification of a new view, a new view is allowed to be defined as a subview of other views (**superviews**).

The view definition used by Bertino [4] has the following format:

create-view Viewname
[**properties** list_of_property_names]
[**view-query** query]
[**additionalproperties** list_of_properties]
[**methods** list_of_method_specifications]
[**superviews** list_of_views]
[**OID** true_or_false]
[**identityfrom** list_of_class_names]

The paper also includes a detailed description of keywords, and extensions of the view mechanism such that schema versions and the possibility of experimenting with schema changes before executing them are supported.

Next, using a portion of a university database shown in Figure 1, we consider several examples on various aspects of constructing views. In order to increase the flexibility of the view mechanism, we extend the view definition such that it allows us to define *parameterized views.*

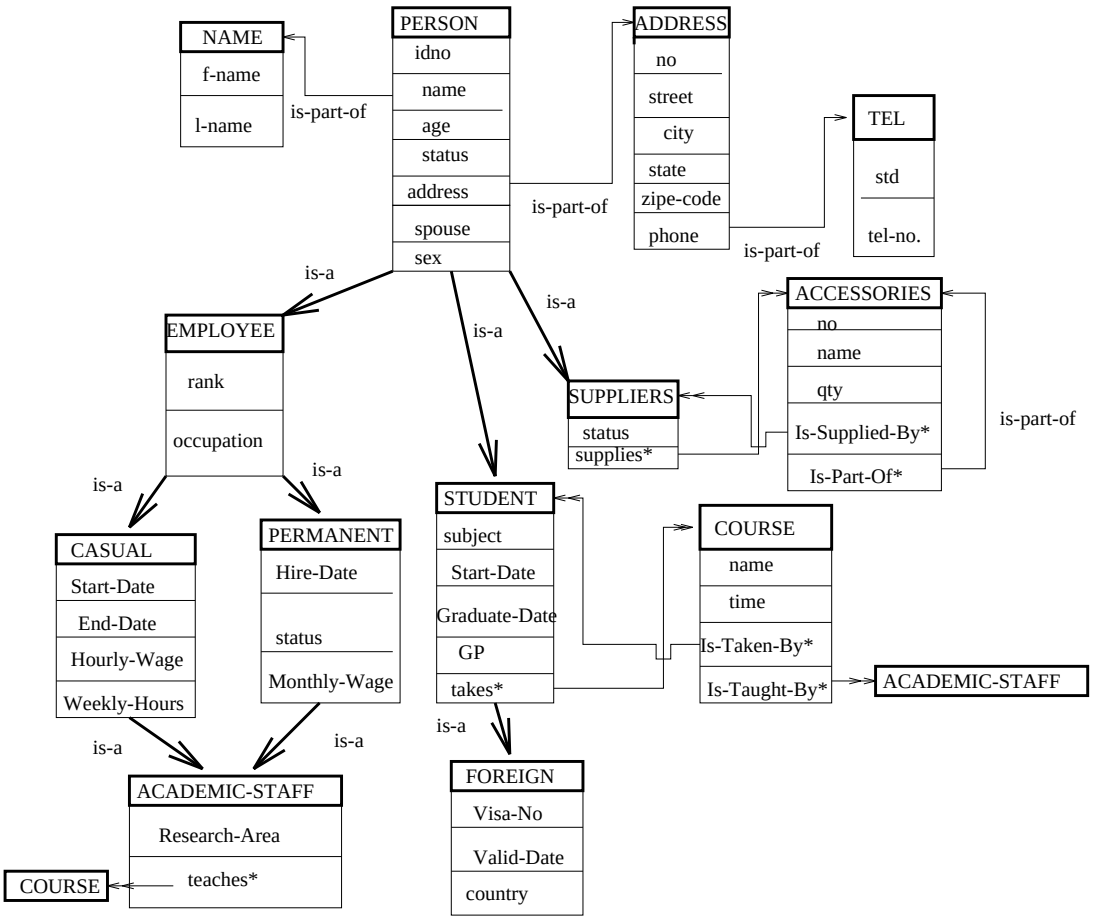

Figure 1: A portion of University Database.

In Figure 1, we use "*" symbol after a property name to denote that the property is multi-valued.

2.2 Examples

In this subsection, we present a number of examples in order to illustrate various aspects of the view model.

As pointed out in [15], a view can be constructed in two distinct ways: *top-down* or *bottom-up*. In the *top-down* approach, large classes are divided into smaller ones via *specialization* (a similar operation in relational systems is to define a view by selecting a subset of tuples from a large table). In the *bottom-up* approach, small classes are combined to form larger classes via *generalization* (the analogous operation in relational systems is to define a view as the union of several tables).

In the following examples, "*" denotes all hierarchical instances of a class. For example, PERSON* denotes all person whatever they are students, employees, suppliers, etc.

2.2.1 Top-down Approach

The following examples illustrate the construction of a view using the top-down approach.

Example 1 *Define separate views for each of the following cases:*
- *persons whose age is less than 21;*
- *persons whose age is between 21 and 95; and*
- *persons whose age is greater than 65.*

One possible way of writing the views is:

create-view Minor
select P
from P:PERSON*
where P.age $<$ 21;

create-view Adult
select P
from P:PERSON*
where P.age $\geq$ 21
and P.age $\leq$ 95;

create-view Senior
select A
from A:Adult
where P.age $>$ 65;

In the above examples, the views *Minor* and *Adult* contain a subset of object instances of th class PERSON, and the view *Senior* contains a subset of objects of the view *Adult* which satisfies the predicates. Furthermore, just instances of class PERSON are visible that satisfy the views' predicate.

The next example illustrates how behavioral views can be defined.

Example 2 *Define views which return junior, minor, and senior students separately.*

In order to declare corresponding views, we first define view *New_Student* which has an additional method *year* that computes a student's year. The computation is based on the attribute *Current_Date* which gives the current date of the system and the property *Start_Date* which indicates the starting date of the student's study.

create-view New_Student
select S
from S:STUDENT*
methods(**numeric** year());

One possible implementation of the method *year* can be as follows:

numeric year()
begin
X: STUDENT;
return (Current_Date - X. Start_Date)
end

Then, the declaration of views are:

create-view Junior_Student
select X
from X:New_Student
where X.year $\leq$ 1;

create-view Minor_Student
select X
from X:New_Student
where X.year $>$ 1
and X.year $<$ 3;

create-view Senior_Student
select X
from X:New_Student
where X.year $\geq$ 3;

Since each one from the above views contains a subset of object instances of the *New_Student,* they are subviews of that. Moreover, because the view *New_Student* is a subclass of the class STUDENT, then the views are subclasses of STUDENT too.

So far we used view mechanism to restrict the set of visible object instances of a class (this is analogous to selection operation in the relation systems). Examples 3 and 4 illustrate another possible usage of view mechanism that restricts properties which are visible; this is similar to the projection operation in the relational systems.

Example 3 *Define a view which returns name, status, and phone number of suppliers who are resident of Sydney, and supply Macintosh personal computers.*

create-view Suppliers_In_Sydney
select S.name, S.status, S.address.phone.std, S.address.phone.tel_no
from S:SUPPLIERS
where S.supplies.name="Macintosh"
and S.address.city="Sydney"
and S.address.state="NSW";

Above view returns indicated properties of supplier's specification, and the set of instances that satisfy the view predicates.

Example 4 *Construct a view which returns the information of postgraduate students who are from Australia. It is required that name, idno, subject, Graduate_Date, and country to be visible.*

One possible declaration of the view is as follows:

create-view Australian_Graduate_Students
select F.idno, F.name, F.Subject, F.Graduate_Date, F.country
from F:FOREIGN
where F.status = "graduate"
and F.country="Australia";

If we want to have a separate roster of postgraduate students from each country, we should redefine the above view for each value of the property *country.*

In order to increase the effectiveness of the view model, we allow *parameterized* view. In such views, parameters are bound to the actual values at the time the view is evaluated. For instance in the above example, we may define a view *Graduate_Students(X)* with the parameter X whose value is determined during query processing. When the value of parameters are determined, corresponding objects are evaluated dynamically.

create-view Graduate_Students(X)
select G
from G:FOREIGN
where G.country=X;

This statement effectively declares many views including *Graduate_Students(Australia)* and *Graduate_Students(Iran)* each with a different name and a different object instances.

Note that although the set of instances of each view, *Suppliers_In_Sydney* and *Graduate_Students*, is a subset of instances of classes SUPPLIERS and FOREIGN, respectively. But because they have fewer properties than their base classes, they can not be the subclasses of SUPPLIERS and FOREIGN (according to the principle of inheritance, a subclass must inherit all properties of superclass(es)).

Example 5 and 6 illustrate the usage of parameterized views in more complex cases.

Example 5 *Define a view that retrieves students who take the course X with the lecturer Y.*

create-view Students_Of_Course_Of_Lecturer(X,Y)
properties (Student_Name)
select C.Is_Taken_By.name
from C:COURSE
where C.name=X
and C.Is_Taught_By.name=Y;

Example 6 *Define a view that lists all employees whose monthly salary are more than X.*

Since an employee is possible to be a casual or a permanent, we should define a method that computes the monthly salary for casual employees. The method *salary* computes the monthly earning for casual employees.

create-view Wage(X)
properties(name,salary)
select E.name
from E:EMPLOYEE*
where E.salary()>X
methods(**numeric** salary());

One possible implementation of the method *salary* is as follows:

numeric salary()
begin N:string; Y: EMPLOYEE*;
N=**classof**(Y);
If(N==“PERMANENT”) **then return** (Y.Monthly_Wage)
else If(N==“CASUAL”) **then return**(4*Y.Weekly_Hours*Y.Hourly_Wage)
end

2.2.2 Bottom-up Approach

Another possibility to construct views is to declare a view such that it contains all instances of a number of base classes (or views). In other words, they are *generalization* of base classes (views) or their superviews. The following examples illustrate the construction of a view on top of existing classes or views.

Example 7 *Define a view that contains all employees, both casual and permanent.*

One possible declaration of such view is as follows:

```
create-view Casual_And_Permanent
select O, P
from O:CASUAL, P:PERMANENT
methods(date Start_Date(), date End_Date(), string status(), numeric Weekly_Wage());
```

The following methods retrieve the required values from the class PERMANENT and CASUAL. For example, the method *Start_Date()* returns the value of property *Hire_Date* from the class PERMANENT or *Start_Date* from the class CASUAL.

```
date Start_Date()
begin
N:string; Y: EMPLOYEE;
N=classof(Y);
If(N=="PERMANENT") then
return (Y.Hire_Date)
else If(N=="CASUAL") then
return(End_Date);
end
```

```
date End_Date()
begin
N:string; Y: EMPLOYEE;
N=classof(Y);
If(N=="PERMANENT") then
return (nullvalue)
else If(N=="CASUAL") then
return(Start_Date);
end
```

```
string status()
begin
N:string; Y: EMPLOYEE;
N=classof(Y);
If(N=="PERMANENT") then
return (Y.status)
else If(N=="CASUAL") then
return("casual");
end
```

```
numeric Weekly_Wage()
begin
N:string; Y: EMPLOYEE;
N=classof(Y);
If(N=="PERMANENT") then
return ((Y.Monthly_Wage/22)*4)
else If(N=="CASUAL") then
return(Y.Weekly_Hours*Y.Hourly_Wage);
end
```

We note that the class EMPLOYEE is a superclass of the view *Casual_And_Permanent* and the classes ACADEMIC_STAFF, CASUAL, and PERMANENT are subclasses of this view. Furthermore, by inserting the view *Casual_And_Permanent* in the middle of the class hierarchy, multiple inheritance in case of ACADEMIC_STAFF is turned into single inheritance EMPLOYEE, *Casual_And_Permanent,* and ACADEMIC_STAFF. This is explicitly demonstrated in the following view definition.

```
create-view Academic_Staff_View
select A
from A:ACADEMIC_STAFF
superviews( Casual_And_Permanent);
```

We may combine the two above mentioned approaches (top-down and bottom-up) of view definition. For example, we can split the information of view *Casual_And_Permanent* into two specialized views and then combine them to find a more general view. The following example illustrates this.

Example 8 *Define three separate views such that the first two views contain software supporters and programmers, respectively and the third one contains both.*

One possible way to define such views is as follows:

```
create-view Software_Supporters
select E
from E: Casual_And_Permanent
where E.occupation="software_supporter";
```

```
create-view Programmers
select E
from E: Casual_And_Permanent
where E.occupation="programmer";
```

The views *Software_Supporters* and *Programmers* contain object instances of the view *Casual_And_Permanent* that are software supporters and programmers, respectively. Then, the following view contains object instances of both views.

create-view Technical_Staff
select P, S
from P:Programmers, S:Software_Supporters;

The result of defining the above views is the extension of the class hierarchy such that the classes EMPLOYEE and PERSON, and the views, *Casual_And_Permanent* and *Technical_Staff*, become the superclasses of the views, *Software_Supporters* and *Programmers.*

It is worth noting that although in some cases, in the above examples, a view hierarchy can be derived for a view, in general this may not be possible. A view derived from a class (or a view) in general may have fewer properties than the class (view) and/or additional methods. Therefore, the view cannot be neither a subclass (subview) of the class (view) from which it is derived (since it has fewer properties) nor a superclass (superview) of it (because it has more methods).

3 Inferring the View Hierarchy

In the previous section, we noted that not only a view hierarchy can be defined for a view (using **superviews**) but also in some cases the view hierarchy can be derived from the view definitions. In this section, we further develop and formalize this idea.

A view defined using *top-down* or *bottom-up* approach can be further sub-divided as follows.

T. Top-down Approach.

Suppose that a view V is derived from a class (or a view) C. Then the view V may satisfy one of the followings:

T1. inherit all properties of C, and the set of its instances be a subset of instances of C (see Examples 1 and 5);

T2. have more properties, and/or methods, than C and the set of its instances be a subset of the instances of C (see Examples 2 and 6);

T3. have more properties and fewer methods (or vice versa) than C; and

T4. have fewer properties and methods than C (see Examples 3 and 4).

B. Bottom-up Approach.

Suppose that the view V is defined over classes (or views) $C_1, \ldots, C_n$. Then one of the following cases may happen.

B1. V inherits all common properties and methods of base classes (or views). It contains all instances of $C_1, \ldots, C_k$, and selected instances of $C_{k+1}, \ldots, C_n$. Note that because, in general case, $C_1, \ldots, C_k$ are not required to have the same properties, then the view V may have fewer properties, and/or methods, than base classes (or views).

B2. $C_1, \ldots, C_k$ have the same properties. V inherits all properties and methods of the base classes (or views). It includes all instances of $C_1, \ldots, C_k$, and selected instances of $C_{k+1}, \ldots, C_n$, (see Example 8).

B3. V contains all instances of $C_1, \ldots, C_k$, and selected instances of $C_{k+1}, \ldots, C_n$. It has more properties, and/or methods, than base classes (or views), (see Example 7).

Based on the above discussion, we can distinguish four types of inheritance relationship among classes (or views) and derived views.

1. **Specialization inheritance.** *Views inherit all the properties of base classes (or views), and they also have some additional properties and/or methods. Examples are cases T2, and B1. In the case T2, the view V is a specialization of C, or in other words V is a subclass (or subview) of C. In the case B1, each C_i is a subview (or specialization) of V for $1 \leq i \leq k$. Moreover, if a class (or view) D is a superclass (superview) of base class(es) (or views) then D is also a superclass (superview) of V.*

2. **Constraint inheritance.** *Views consist of all object instances of base class(es) (or views) that satisfy given constraints (cases T1, and B2). In other words, every object of the view is also an object of the base class(es) (or vice versa). Hence, the view is a subview of base class (or vice versa). For example in case T1, because every object of V is an object of C, V is a subview of C. In case B2, since every object C_i $(1 \leq i \leq k)$ is an object of V, C_i $(1 \leq i \leq k)$ is a subview of V.*

3. **Strict constraint inheritance.** *In this case views not only consist of all object instances of base classes (or view) that satisfy given constraint, but their properties are fewer than the base classes (or views). In other words, not only the view filters out the object instances of base classes (or views) but also projects the properties (see cases T4 and B3). V is a strict subview of C (case T4). In the case B3, C_i $(1 \leq i \leq k)$ is a strict subview of V.*

4. **Proper specialization inheritance.** *Here, not only derived views filter out the object instances of the base classes and project the properties, but they also contains new additional properties or methods (see the case T3). Views consist of all object instances of base classes (or views) that satisfy given constraint. They do not inherit all properties or methods of base classes (which contradicts the principle that a subclass must inherit all properties and methods of superclasses). They have new additional properties or methods. Therefore, derived views cannot be specialization of the base classes because they do not inherit all properties of base classes. Moreover, they cannot be strict subviews of base classes since they have additional properties and/or methods. Hence, the view V is a proper specialization of C.*

So far we have discussed usage of the view mechanism as *user shorthand,* and *schema changes.* A view may also be used for *protection* purposes. In the next section, we describe the usage of views for controlling access in an authorization system.

4 Authorization System

As described in the previous section, view definitions use predicates on the values of properties of a class, and/or combine the object value of several classes to access objects that meet the specified conditions. If authorization is granted on a view, the access is restricted to the object instances that are not filtered out by the view. Hence, views provide a powerful mechanism for authorization based on database contents. They are able to be used to provide the desired levels authorization granularity (class, object, instance-variable, and method-based). Examples 9-12 show how a view model can be used to provide different levels of granularity.

Example 9 *Define a view such that contains all information related to all students (class-based authorization).*

create-view Students_Info
select S
from S: STUDENT;

Example 10 *Define a view such that contains information related to a particular student. let's say student with idno 9272860 (object-based authorization).*

create-view Student_9272860
select S
from S: STUDENT
where S.idno=9272860;

Example 11 *Define a view such that contains name, subject, and Start_Date related to all students (instance-variable based authorization).*

create-view Students_Name_Subject
select S.name, S.subject, S.Start_Date
from S: STUDENT;

Example 12 *Define a view such that contains method associated with class STUDENT which obtains the greatest point (GP) (method-based authorization).*

Suppose that the method *GP_Computing* computes the associated value with the instance variable *GP related to each student.* Then, it is possible to define a view *Computing_GP* as follows:

create-view Computing_GP
methods GP_Computing from STUDENT;

The view *Computing_GP* can access only the method *GP_Computing.* If a user has *execute* authorization on the view *Computing_GP,* (s)he can only execute the method associated with the view. Therefore a view model provides the facility that different interfaces for a class can be defined and can be used as units of authorizations.

Furthermore, views provide a useful mechanism for efficient storage usage in an authorization system based on object instances. This is true because instead of appending a user entity to all object instances which the user is authorized, a view can be defined that includes all object instances associated with the user and then authorizing him/her to access the view. In other words, view mechanism provides a facility to group objects which share the same access control list. The maintenance of authorizations becomes easier.

The authorization rule in a database with views, can be formally defined as follows.

Definition 1 *An authorization rule is a triple* (u, v, r) *where*
$u \in U$, *U is the set of users/roles in the system;*
$v \in V$, *V is the set of views in the system; and*
$r \in R$, *R is the set of access rights.*

The triple (u, v, r) states that the user/role u has access right r on the view v.

In order to discuss the use of the views in object-oriented databases that enforce discretionary access control, we need to answer questions such as who is a user?, what is the unit of authorization?, what is the set of access rights?, who can grant/revoke these rights?, which security requirements must be considered?, how to enforce these security requirements?, etc..

To answer these questions, we first introduce the notation *access view* which consists of the ordinary view specification together with security specification (we only consider discretionary access control). Later we use access views to give a solution to the above questions, and then consider the effect of hierarchical structure of each domain of authorization rule (user, view, access right).

Due to the fact that satisfaction of the conditions in the authorizations depends on the values of the properties of objects and can therefore change over time. Hence, to determine the consistency of the authorizations when an authorization system which supports positive and negative authorizations is difficult. For example, consider a user u has *positive read* authorization on the view v which contains postgraduate students, and has *negative read* authorization on the

view w that consists of academic staff information. If no common objects exist in v and w (an entity is a member of postgraduate students and is a member of academic staff too), then the authorization status is consistent. otherwise it is inconsistent. Therefore, for the rest of the paper we assume that The proposed authorization system uses *positive authorization* policy which requires explicit specification of allowable operation, and hence the lack of authorization for a user to a view implies no access to this view. We denote by u a user/role identity, v a view, r an access right, t an authorization type, op an operation, V the set of views, U the set of users, and R the set of access rights.

4.1 Access Views

In order to enforce security requirements and to protect an object against unauthorized accesses, the authorization system has to know the exact users' access rights. This can be accomplished by creation of access views. An *access view* is an extended view that in addition to the view specification, has an authorization list for discretionary access control.

Definition 2 *An access view is an extended view that includes view's specification and a discretionary access control list. The format of the access view is as follows:*

> **access-view** *Viewname[parameters]*
> *– view specification as described in Subsection 2.1*
> **Auth-Spec** *authorization_information*

where Viewname is the name of the access view, and must be different from the names of other access views and classes; **Auth-Spec** *provides specification of authorization information which is an aggregation of the* **ownid** *and* **ACL***;* **ownid** *is the identity of the user that owns the view and* **ACL** *is the access control list for the view.*

Two points is worth mentioning. Firstly, when a *parameterized access view* is defined, parameters are bounded to the actual values at the time the view is evaluated. Secondly the authorization specifications can be read by users who have access to the view but it can only modified by the owner of the view, or users who have grant or revoke authorization.

In general, metadata of a database system in a view model consist of two classes of data. Firstly, conceptual data are mapped to the stored data. In case of object-oriented databases, these are called classes which are mapped to stored objects. Each object is an instance of a class. Secondly, views are declared (or created) based on the classes or pre-defined views. Therefore, ther can be two classes of users: those who are allowed to access/manipulate classes, and those who are only authorized to access objects through views.

Authorization Rule 1 *There are two classes of users: security officers, and normal users.*

Security officers have the supervision on the entire database activities and are also responsible for creating views based on classes. Furthermore, they are responsible to create classes and to perform schema changes.

Normal users can only access data through access views.

Users may want to create views as subviews of other views or as superviews of other views. They can create new views based on a predefined view if they have *create-view* privilege, or if they are the owners of the base views.

Authorization Rule 2 *A user can create an access view over other views if the user owns the views, or has a create-view authorization on base views.*

When a user creates a view, (s)he becomes its owner. This is specified by the **ownid**. The owner has full authorization to the view, and may determine who can access it and how.

Authorization Rule 3 *The creator of a view is its owner. The owner of the view specifies users' access rights fro any view operations and for only view-instance operation that (s)he is authorized.*

We assume that there are two set of operations: *view operations,* and *view-instance operations. View operations* are used to manipulate the view definitions and typically are: *delete-view, create-view, modify-view, grant,* and *revoke. View-instance operations* are used to access/manipulate the instances of a view. A typical collection of operation is: *read-definition, read, write,* and *delete* (see Section 5.2).

Authorization Rule 4 *The ownership is transferable to other users. However, a view has a unique owner at any time.*

The authorization system must check the eligibility of users and their privileges on the view before any operation. Hence for each view, an **ACL** is defined and lists who and how can access the view.

Definition 3 *An access control list (ACL) of a view v is a list of pairs (u, r) where u is the identity of the user and r is the access right granted to the view v.*

As stated in [4], views (consequently access views) can have additional properties whose types can be pre-defined views or classes (composite view). Also we may have a query whose scope of access can be a view (class) and its subviews (subclasses). In such cases, it is essential to determine whether an authorization right to the root of a view is to be extended to all its constituents or not.

Rabitti Bertino, Kim, and Woelk [6] have distinguished two types of authorization for accessing composite objects and class-hierarchy objects, called *full* and *partial* authorizations.

A *full authorization* on a composite (or class-hierarchy) object implies the same authorization on each component of the composite object (or on each class of the hierarchy). However, a *partial authorization* does not extend to the descendants of the composite (or class-hierarchy) object. We here employ the same concept.

Definition 4 *An access right r has a form of (t, op) where t denotes the type of authorization right, and op indicates the operation such as read-definition, read, write, delete, execute, create-view, modify-view, delete-view, grant, and revoke.*

Therefore, an authorization rule $(u, v, (t, op))$ states that the user u is authorized to execute operation op of type t on the view v.

Because there are three types of hierarchy, *user-role hierarchy, view-hierarchy,* and *composite hierarchy (or object),* it is required to determine whether an authorization right to a root node must be extended to its descendants or not. To determine the authorization type t, two approaches may be taken. First approach is that *full (F)* or *partial (P),* be applied to all three types of hierarchy. This approach is simple but is not flexible. Second approach is to determine authorization type t for each hierarchy separately. In the latter case, the authorization type t will become a triple such that each element determines the associated authorization type to one of the hierarchies. We have chosen the second approach for the rest of this paper.

Definition 5 *An authorization type t is a triple (t_1, t_2, t_3) where each t_i $(1 \leq i \leq 3)$ indicates the type of authorization of op for user-role hierarchy, view-hierarchy, and composite hierarchy, respectively; and t_i is either F (full) or P (partial).*

We assume that partial (P) authorization right for each type of hierarchy will become effective unless full (F) authorization right is explicitly specified. The definition of the user-role hierarchy and the usage of authorization type along it are discussed in the next section.

Example 13 *Let user u_1 create a view Adult (see Example 1 in Section 2.2) and gives access rights ((P, F, F), read), write, and ((, F ,P), read) to users u_2, u_3, and u_4, respectively.*

Then, the access view *Adult* will look like this:

access-view Adult
select P
from P:PERSON*
where P.age ≥ 21
and P.age ≤ 95
Auth-Spec.ownid u_1
Auth-Spec.ACL $\{(u_2, ((P, F, F), read)), (u_3, ((P, P, P), write)), (u_4, ((P, F, P), read))\}$;

The view *Adult* is declared on the composite class PERSON (the components of the class PERSON are classes NAME and ADDRESS; and the component of the class ADDRESS is the class TEL). Therefore, the *full read* authorization for the user u_2 implies that the user u_2 not only can read the view *Adult* and consequently the class PERSON, but he/she can also read the components of the class PERSON (ADDRESS, NAME, and TEL). However, the users u_3 and u_4 can only access the view *Adult* and consequently the class PERSON. Moreover as indicated by "*" in the view definition, the view population might be all hierarchical instances of the class PERSON. So the full authorization right on the view implies the same right on all instances of its subclasses (or subviews). As a result, view population includes all hierarchical instances. Otherwise, it just includes the object instances of the root of the hierarchy. For instance, users u_2 and u_4 can read all constrained instances of the class PERSON including students, employees, suppliers, etc. However, the user u_3 can only access constrained instances of the class PERSON.

Next, we look at impact of hierarchical structure of domains of the authorization rule (u, v, r) on the authorization system. We propose rules to derive implicit authorization from explicit ones.

5 Implication Rules

In this section, we employ the hierarchical structure (or order) of each domains of authorization rules (u, v, r) to provide deductive rules which improve the efficiency of the authorization system.

5.1 Authorization Users

To reduce the number of explicit authorization rules in the system, one possibility is that users be grouped according to their roles. The roles are then organized in some hierarchical structure [6, 14]. Therefore based on the principles of generalization and specialization (because a higher-level roles will inherit accesses given to more general (or lower-level) roles), explicit inclusion of the same authorization rule for more special (or upper-level) roles can be removed. In many situation, a natural hierarchy of roles exists. An example is shown in Figure 2.

A node of the graph represents a role, and a directed arc from one role to another indicates that the authorizations for the higher-level role subsume the authorizations for the lower-level role. For example, the authorization rights for the role *Personal-manager* subsume the authorization rights of the roles *Academic-manager, Staff-manager, Academic_clerks, Personal_clerks,* and *Employee* (see [6] for formal definition of role hierarchy). Consider each employee of the role *Employee* is authorized to read their personal object which is defined by the view *Personal_Info* where consists of *idno, name, address, age,* and *spouse.*

access-view Personal_Info(Current_idno)
select P.idno, P.name, P.address, P.age, P.spouse
from P:PERSON*
where P.idno = Current_idno
Auth-Spec.ownid u_1
Auth-Spec.ACL $\{(Employee, ((P, P, F), read))\}$;

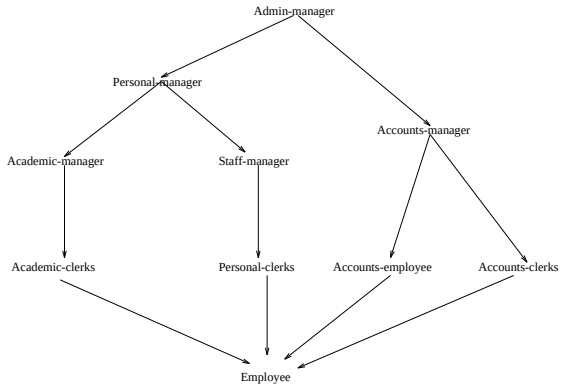

Figure 2: A sample user role hierarchy.

Having the explicit authorization rule *(Employee, ((P, P, F), read))* implies that all members of the role *Employee* can read *Personal_Info.* It is clear that all upper-level roles must be allowed to read the view. If the explicit authorization is the only way to allow users to access data, then the *read* authorization must be replicated for all upper-level roles. The existence of many such authorizations may lead to inefficiency. Therefore, this is a good idea that upper-level roles are allowed to have all rights associated with low-level roles. Note that the value of the *Current_idno* when the view is evaluated must be specified, and is an identity number of an employee. To derive implicit authorization for each role, we have the following rule:

Implication Rule 1 *Explicit authorization for a role results in implicit authorizations for all higher-level (upper) roles corresponding to it.*

Therefore, all upper-level roles, *Academic-clerks, Personal-clerks, Accounts-clerks, Accounts-employee, Academic_manager,* etc., have *read* right on the *Personal-Info* implicitly, by the above rule.

Because a role contains a set of users, it is then possible to give access right on a view v to a role, rather than giving it to all members of the role individually.

Definition 6 *A value u in authorization rule (u, v, r) is either a user identity or a role identity. If the role identity is determined, all users of the role u will have the right r on the view v. Otherwise the specific user has the right.*

Suppose the following view that contains information about casual employees' payments is defined. The required information are *idno, name, rank, Hourly_Wage, Weekly_Hours,* and *Weekly_Wage.*

access-view Casual_Payment(Current_idno)
select C.idno, C.name, C.rank, C.Hourly_Wage, C.Weekly_Hours, Weekly_Wage
from C:CASUAL
where C.idno = Current_idno
methods(**numeric** Weekly_Wage())
Auth-Spec.ownid u_1
Auth-Spec.ACL $\{(Account_employee, ((P, P, P), read))\}$;

where the method *Weekly_Wage* computes the weekly payment of an employee.

read authorization for the role *Account_employee* means that members of roles *Account_employee, Account_manager,* and *Admin_manager* can only access the information. Suppose we want that employees are themselves allowed to access their own information too. If we grant *read* authorization to the role *Employee* (as a matter of fact the tuple *(Employee, ((P,P,P),read))* is added to **Auth-Spec.ACL**), then all upper-level roles can also access it by Implication Rule 1. The

only way to solve this is to grant *read* authorization to all individual casual employees. If a denial (or negative) *read* authorization for not authorized roles is specified, conflicting authorizations will arise.

However, in our model, the role identity is given, and the first element of the authorization type t will determine the type of authorization associated to the role hierarchy. Therefore, by assigning *full read* authorization for the role *Account_employee ((Account_employee, ((F, P, P), read)))*, the same right will be implied for the role *Employee*.

Implication Rule 2 *Explicit full authorization for a role results in implicit authorization for all associated lower-level roles.*

For example, if the *Academic_manager* has *full read* right on a view v (v can be considered to contain research area and grants associated with the academic staff) Then, all descendant roles of the *Academic_manager* such as *Academic_clerks*, and academic employees will implicitly have the same authorization on v.

We assume that partial authorization right for roles will become effective unless full authorization right is explicitly specified.

5.2 Authorization Operations

As observed, an access right r in our authorization system is of the form (t, op) where t denotes the authorization type: *full* or *partial* and *op* denotes the operation. So we first define allowable operations (access rights) and then consider the effect of authorization type t on the authorization rules.

We assume that two sets of system-defined methods (called operations) are provided by an object-oriented database system (these methods are implemented by the system in an efficient way). The two sets are *view operations*, and *view-instance operations*. We assume that they consist of:

1. *View operations: create-view, modify-view, delete-view, grant, revoke.*
2. *View-instance operations: delete, write, execute, read, read-definition.*

A *create-view* operation is used to create views over views. A *modify-view* operation is used to change the definition of a view. A *delete-view* operation is used to delete the definition of a view. *grant* and *revoke* operations are used to grant and revoke both the view operations, and view-instance operations to and from users, respectively. The *grant* and *revoke* authorizations cannot be propagated. This means that only the owner of a view is allowed to give the privileges *grant* and *revoke* to other users. *read* and *read-definition* operations are used to read the instances of a view and read the definition of a view, respectively. An *execute* operation is used to perform the methods associated with a view. In other words, the *execute* operation can be considered as an *invoker* that can invoke methods associated with a view. A *write* operation is used to modify and/or to create an instance of a view. A *delete* operation is used to delete an object instance of a view.

Definition 7 *The set of operations includes read-definition, read, write, delete, execute, create-view, delete-view, modify-view, grant, revoke. We assume the following order among these operations:*

create-view $>$ *read-definition*
modify-view $>$ *delete-view* $>$ *read-definition*
grant $>$ *revoke*
execute $>$ *write* $>$ *read* $>$ *read-definition*
delete $>$ *read* $>$ *read-definition*

This means that the holder of an operation of a higher order possesses operations of the lower order. For instance having access right *execute* implies that the user has rights *write, read* and *read-definition* because the user must be able to read the values and definitions associated with the parameters of a method and must be able to write the returned result in order to execute a method. To delete an object instance to be deleted, a user must first access it, and later remove it. This implies *read* and *read-definition* authorizations. The access right *modify-view* implies that the rights *delete-view,* and *read-definition* are also allowed. The operations' hierarchy is shown in Figure 3.

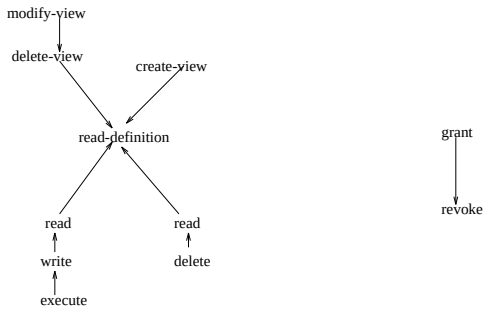

Figure 3: A sample operation hierarchy.

Using this hierarchy and the following rule, it is sufficient to specify explicit authorization only for the highest operation.

Implication Rule 3 *If an authorization is given for an operation in the operations' hierarchy, then this implies the authorization of all operations below it.*

For example, the authorization rule $(u, v, (t, execute))$, authorizes the user u to use operations *write, read,* and *read-definition* on v.

Implication Rule 4 *The full authorization for an operation implies the partial authorization for the same operation.*

5.3 Authorization Views

In the Section 3 four types of inheritance, *constraint inheritance, strict constraint inheritance, specialization inheritance,* and *proper specialization inheritance* among views (classes) and derived views were introduced

In general if the base views contain more information than the derived views, it is reasonable to assume that a user u has the same authorization right r on the derived view v if the user has right on the base view w (cases *constraint inheritance* and *proper specialization inheritance).*

However, to preserve the view's privacy in the case of the *specialization inheritance* and *proper specialization inheritance* which a derived view contains information in addition to base views, an authorization right on superview should not imply the same authorization on the derived view except when the full authorization for the superview is indicated. The following two rules express that.

Implication Rule 5 *If a view v relationship to view w is a constraint inheritance or a strict constraint inheritance, then explicit access right on view w results the same access right implicitly on the view v.*

For example, let a user have *read* right on the view *Technical_Staff* (see Example **??**). Then since views *Software_Supporters* and *Programmers* have *constraint inheritance* relationship with the view *Technical_Staff,* the user will have the same right on the latter two views.

Implication Rule 6 *If a view v relationship to a view w is a (proper or) specialization inheritance or is component of, then explicit full authorization on the view w results the same access right implicitly on the view v.*

For example, the role *Employee* which has full read authorization on the view (see Example **??**). When the view is materialized, the components of the class *PERSON (ADDRESS* and *NAME)* are authorized for retrieval too.

6 Access Control

The authorization system determines whether a user's access request to access views should be allowed or not.

In the proposed system, all user accesses to the database are controlled by a set of *access views, (AV).*

Definition 8 *An AV is the set of all access views. AV may only be accessed or manipulated by the authorization system with the following commands.*
For each view $v \in AV$, we have
grant(u, v, r) *-adds the pair (u, r) to the ACL of the view v.*
revoke(u, v, r) *-removes the (u, r) from the ACL of the view v.*
Own(v) *-retrieves the owner's identity of the view v.*
change-own(v) *-changes the owner of the view v.*
Accesslist(v) *-retrieves the authorization list associated with the view v.*

The only possible way for all users (except security officers) to access data in the database is via AV.

Definition 9 *An access request is a triple (u, v, r) where $u \in U$ is a user who requires the access right $r \in R$ to the view $v \in AV$.*

Definition 10 *An access request (u, v, r), $v \in AV$, $u \in U$, $r \in R$ is valid, if the view v is in AV, and the pair (u, r) exists in the ACL of the view v or can be derived by applying one of the implication rules.*

Every access request is checked against the AV to determine whether it is authorized or not. More specifically, an access request (u, v, r) is authorized if and only if the following function returns true.

function Access(u, v, r)
begin
If u= **Own**$(v) \bigvee (u, r) \in$ **Accesslist**(v)
  then return true
  else If Implicit_Access(u, v, r)
    then return true
    else return false;
end

function Implicit_Access$(u, v, r))$
begin
/* checking access request against user authorization and operation hierarchies */
If $\forall\ u_j$ role of the user $u\ \exists(u_j, r) \in$ **Accesslist**$(v) \bigvee$
  $\forall\ u_j$ descendant of the role of the user $u\ \exists(u_j, r) \in$ **Accesslist**$(v) \bigvee$

$\forall u_j$ ancestor of the role of the user u $\exists(u_j,((F,,),op)) \in$ **Accesslist**(v) $\bigvee$

$\forall$ op_j higher-level of the operation op $\exists(u,(t,op_j)) \in$ **Accesslist**(v)

then return true

/* checking access request against view authorization hierarchy */

else If $\exists w$ that v is (strict or) constraint inheritance of w $\bigwedge \exists(u,r) \in$ **Accesslist**(w)

then return true

else If $\exists w$ that v is (proper or) specialization inheritance of w

$\bigwedge \exists(u,r) \in$ **Accesslist**(w) $\bigwedge t_2 = F$

then return true

else If $\exists w$ that v is component of w $\bigwedge \exists(u,r) \in$ **Accesslist**(w) $\bigwedge t_3 = F$

then return true

else return false;

end

Where denote by $\forall$ for all, $\bigvee$ logical operation or, $\exists$ exist, and $\bigwedge$ logical operation and.

7 Summary and Future Work

The main purpose of this paper is to discuss the use of views in Object-Oriented database systems for discretionary access control. We have emphasized problems of context and content-dependent authorizations. We have discussed application of views in object-oriented databases, in particular for data protection and their effect on class hierarchy in the context of the view model proposed by the Bertino [4].

We have introduced *parameterized views* in order to increase the flexibility of the view model, and showed how in special cases a view hierarchy can be inferred from the definitions of views. Four possibilities of inheritance among views were discussed, *(strict) constraint inheritance* and *(proper) specialization inheritance.*

We have also defined *access views* as the mechanism for access control. We have discussed discretionary security requirements for authorization systems and provided some rules for computing implicit authorization from explicitly defined ones along with the three authorization dimensions, users, operations, and views.

Finally, we have presented the form of a valid access request and how the validity of the request can be verified.

Our future work is to provide a multilevel authorization model for object-oriented databases that addresses the issues of context and content-dependent classification, dynamic classification, inference, and aggregation.

References

[1] A. Baraani-Dastjerdi and J. R. Getta and J. Pieprzyk and R. Safavi-Naini. A Cryptographic Solution to Discretionary Access Control in Structurally Object-Oriented Databases. In *Proceedings of the 6th Australian Database Conference (ADC'95)*, volume 17(2), pages 36–45. Australian Computer Science Communications, Ron Sacks and Justin Zobel (Editors), January 1995.

[2] D. B. Faatz and D. L. Spooner. Discretionary Access Control in Object-Oriented Engineering Database Systems. In *Database Security IV, S. Jajodia, and C.E. Lanwehr (Editors)*, pages 73–83. Elsevier Science Publishers B. V. (North-Holland) IFIP, 1991.

[3] E. Bertino and H. Weigand. An Approach to Authorization Modelling in Object-Oriented Database Systems. In *Data & Knowledge Engineering, P. P. Chen and R. P. Van de Riet (Editors)*, pages 1–29. Elsevier Science Publishers B. V. (North-Holland), 1994.

[4] Elisa Bertino. A View Mechanism for Object-Oriented Databases. In *Proceedings 3rd International Conference on Extending Data Base Technology (EDBT), Vienna, Austria*, volume 580, pages 136–151. Springer-Verlag, Lecture Note in Computer Science, March 1992.

[5] Elisa Bertino. Data Hiding and Security in Object-Oriented Databases. In *Proceedings of the Eight International Conference on Data Engineering Edited by F. Golshani*, pages 338–347. IEEE Computer Society Press, 1992.

[6] F. Rabitti and E. Bertino and W. Kim and D. Woelk. A Model of Authorization for Next-Generation Database Systems. *ACM Transactions on Database Systems*, 16(1):88–131, March 1991.

[7] J. K. Millen and T. F. Lunt. Security for Object-Oriented Database Systems. In *Proceedings of IEEE computer Society Symposium on Research in Security and Privacy, Oakland, CA.*, pages 260–272. IEEE, May 1992.

[8] Joel Richardson and Peter Schwarz and Luis-Felipe Cabrera. CACL: Efficient Fine-Grained Protection for Objects. In *OOPSLA'92: Seventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 27(10), pages 263–275. ACM SIGPLAN Notices, October 1992.

[9] K. R. Dittrich and M. Hartig and H. Pfefferle. Discretionary Access Control In Structurally Object-Oriented Database Systems. In *Database Security II: Status and Prospects, C. E. Landwehr (Editor)*, pages 105–121. Elsevier Science Publishers B. V. (North-Holland), IFIP, 1989.

[10] M. B. Thuraisingham. Mandatory Security in object-oriented database Systems. In *Proceedings International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 203–210. New Orleans, October 1989.

[11] M. M. Larrondo-Petrie and E. Guides and H. Song and E. B. Fernandez. Security Policies In Object-Oriented Databases. In *Database Security III, D. L. Spooner and Landwehr (Editors)*, pages 257–269. Elsevier Science Publishers B. V. (North-Holland) IFIP, 1990.

[12] M. S. Olivier and S. H. Von Solms. A Taxonomy for Secure Object-Oriented Databases. *ACM Transactions on Database Systems*, 19(1):3–46, March 1993.

[13] R. Ahad and J. davis and S. Gower and P. Lyngbaek and A. Marynowski and E. Onuegbe. Supporting Access Control in an Object-Oriented Database Language. In *Proceedings of the 3rd International Conference on Extyending Database Technology, EDBT'92, Vienna*, volume 580, pages 184–200. Springer-Verlag LCN in Computer Science, March 1992.

[14] R. W. Baldwin. Naming and Grouping Privileges to Simplify Security Management Databases. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 116–132. IEEE Computer Society, 1990.

[15] S. Abiteboul and A. Bonner. Objects and Views. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, J. Clifford and R. King (Editors)*, pages 238–247. ACM SIGMOD, 1991.

[16] S. Heiler and S. Zdonik. Object Views: Extending the Vision. *Proceedings 6th Data Engineering Confrence, IEEE Computer Society Press*, pages 86–93, 1990.

[17] S. Jajodia and B. Kogan. Integrating an Object-Oriented Data Model with Multilevel Security. *IEEE*, pages 76–85, 1990.

[18] T. F. Keefe and W. T. Tsai and M. B. Thuraisingham. A Multilevel Security Model For Object-Oriented Systems. In *Proceeding of the 11th National Computer Security Conference*, pages 1–9. Baltimore, Maryland, October 1988.

[19] T. F. Lunt and E. B. Fernandez. Database Security. *SIGMOD RECORD*, 19(4):90–97, December 1990.

[20] U. Dayal. Queries and views in an Object-Oriented Data Model. *International Workshop on Data Base Programming Languages*, 2, 1989.