A Security Model for Multi-level Object-Oriented Databases Based on Views

Ahmad Baraani-Dastjerdi *Josef Pieprzyk*
Reihaneh Safavi-Naini

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

February 5, 1996

Abstract

In this paper, we employ the view model given by Bertino to propose a new design approach for a secure multi-level object-oriented database system. The central idea is to provide the user with a *multi-level view* derived from a single-level secure object-oriented database. Hence the database operations performed on the multi-level views are decomposed into a set of operations on the single-level objects which can be implemented on any conventional mandatory security kernel.

We show that this approach allows us to overcome the difficulties of handling content and context dependent classification, dynamic classification, and aggregation and inference problems in multi-level object-oriented databases.

Keywords: Object-oriented databases, View mechanism, Multi-level view model, Multilevel secure databases, Security models

1 Introduction

For many years, many database security researches have been done to extend the classical relational model to obtain multilevel relations (see for example [6, 9, 10, 13, 29]). As a result of such work, multilevel relational databases have been developed not only as prototypes but also as products [26, 21].

In recent times several secure models for object-oriented database are proposed. Some proposals consider *single-level object.* Every object is assigned a unique security level that applies to entire object [23, 28]. This approach is attractive for its simplicity and its compatibility with a security kernel. Moreover we do not have to handle the multilevel update problem. [1] However in the real world, there are situations that it is necessary to classify each instance variables of an object at the different security level. In the other hand, the security model has to support *multilevel object.* Hence, there are also other proposals that introduce a finer grain of classification in assigning a security level to each instance variable of an object [20, 24, 16, 18]. However, this

[1] Multilevel objects present the *multilevel update problem [28]: suppose that some instance variables are classified higher than other instance variables of the same object. The problem to be answered is at what clearance level must a subject be to update the object. If the subject is at high level, and writes variables with a lower security, then the subject has "written down" possibly compromising security. If the subject is at lower level, it is not authorized to access the more secure instance variables.*

approach allows us to represent multilevel security to object-oriented databases, but it requires trusted enforcement mechanisms in the object layer.

In order to use the security kernel compatibility of the single-level object and to overwhelm the difficulties of multilevel object, one possible solution proposed by some researchers is to design the schema in such a way that various security constraints which classify instance variables could be handle [23, 28]. This means that for example, if we want the GAP instance variable of the class $STUDENT$ [2] to be secret, we shall create a class $STUDENT_GAP$ with security level SECRET to be a subclass of the class $STUDENT$ (see Figure 2 in Appendix B). Or if the security level of instance variable *address* depends on the value of instance variable *occupation* of the class $EMPLOYEE$, i.e, the security level of *address* is secret if the employee is a chancellor or vice-chancellor; otherwise it is unclassified, then we shall create two classes $S_ADDRESS$ and $U_ADDRESS$ to be subclasses of $EMPLOYEE$ such that either of them contains related addresses to secret or unclassified employees, respectively (see Figure 3 in Appendix B).

There are several problems with this approach. Firstly, if the value of instance variable *occupation* has been changed dynamically, it will be to handle the schema evolution caused by that. Secondly, certain data might end up in unexpected locations and be inaccessible to authorized users. For example, if a secret address object was created as an instance of $U_ADDRESS$ by a secret user, no reference to it would appear in the instance variables in the $U_ADDRESS$ object. As a result, unclassified users would still be unaware of it, and secret subjects would fail to find it in the expected place under the secret subclass $S_ADDRESS$.

There are several advantages with view approaches:

1. View definitions can be regarded as subclasses or a superclasses relating to the base classes (actually virtual subclasses or superclasses) [2, 1]. They therefore provide the possibility of dynamically modifying the database schema but yet retaining its older versions, and provide a means of handling various security constraints which classify properties of a class.

2. A view may be defined on an arbitrary sets of classes and views with variety security levels. We call that *multi-level view.* So by defining a multi-level view for unclassified and secret users, the possibility of storing certain data in unexpected location which is inaccessible to authorized users is eliminated.

3. A view definition can also be regarded as a constraint relating derived data to other data that stored or derived which could be used to handle inference and aggregation problems.

4. Finally, view definitions represent invariants about the underlying data. The data could change, but it would be unnecessary to reclassify any data because views could capture the invariant classification rules.

Hence, we think that developing a *multi-level view model* is provided exactly the right basis for multi-level object-oriented databases and for addressing the content and context-dependent classification, dynamic classification, inference and aggregation problems.

The main objective of this paper is to show how to use the concept of a view model to implement a multilevel security policy on top of a single-level object-oriented database. Hence, we first discuss the essential features of a view model, and then extend the view model to incorporate mandatory label-based security. We continue to show how multilevel view, content and context-classification, and dynamic classification can be supported by this model. Finally we describe how aggregation and inference problems may be handled.

The remainder of this paper is organized as follows. In Section 2, we will describe some concept of multilevel secure databases. In Section 3, we will give an overview of the view model, and then we will present some extensions to the view model in order to incorporate mandatory

[2] The specification of STUDENT and EMPLOYEE have been shown in Figure 1 in Appendix A

security properties. In Section 4, we develop a security model for object-oriented databases based on object views. Handling aggregation and inference problems are addressed in Section 5. Polyinstantiation is discussed in Section 6. Finally, Section 8 concludes the paper.

2 A Brief Concept of Multilevel secure Databases

A multilevel secure database contains information of varying security (or sensitivity). The security levels may be assigned to the data depending on content, context, aggregation, and time. An effective security policy for multilevel security databases must ensure that users only acquire the information to which they are cleared to access. To fulfill this requirement, subjects and objects are assigned security attributes which subjects' attributes are termed *clearance levels* while the objects' attributes are termed *security (or sensitivity or classification) levels.*[3] Subjects are not allowed to modify these attributes and/or their values. Such modification is a system function. Moreover the set of security and clearance levels form a partially ordered lattice with ordering relation "$\geq$"; for example, UNCLASSIFIED $<$ CONFIDENTIAL $<$ SECRET $<$ TOP-SECRET. For security level L_1 and L_2 , $L_1 \geq L_2$ means that security level L_1 *dominates* security level L_2 (if $L_1 > L_2$, it is called that L_1 *strictly dominates* L_2).

In addition, these may be extended to include non-hierarchical cases that incorporate *need-to-know* requirements. For example, we may have a security level named [SECRET,ASIACRYPT] that allows only users with clearance SECRET and member of ASIACRYPT to be allowed access. In other words, all SECRET users are not allowed access. Such division as ASIACRYPT are termed *compartments (or categories).*

Although, in object-oriented databases, the normal encapsulation feature of object-orientation, combined with security labels, provides very natural protection for an object. However if data are not encapsulated in an object which only reference to it appears in the instance variables in other objects, but rather stored as a normal real number, the value would have no natural protection once it leaves the object. A similar problem occurs anywhere an object provides methods that return values of instance variables, when the value leaves the protection of the encapsulated object, security may be compromised. In order to ensures that the security will not be compromised, a multilevel secure model must impose some restrictions based on *flow of information.* To do so, a number of models have been proposed, the earliest and the best known being the Bell-LaPadula model [3].

The Bell-LaPadula model is based on two properties: the *simple security property* and the **-property.* According to the simple property, a subject is allowed to read information from an entity if the clearance level of the subject dominates that of the entity. According to *-property, a subject has write access to an object if the subject's clearance level is dominated by the security level of the object. Informally, a subject cannot *read-up* (simple security property) and cannot *write-down* (*-property).

This security model is not sufficient to ensure multilevel security in a database system because users are able to pose multilevel queries and infer unauthorized information from the legitimate responses that they receive. Furthermore as we said, a security model for object-oriented databases must also take into consideration the information flows that occur as a result of inheritance and the message-passing. Therefore a number of extensions to the Bell-LaPadula security model have been proposed for multilevel security databases [20, 10, 13, 23, 29, 28, 18, 16, 24].

3 View Model

Views in an object-oriented database model have been considered in a number of papers [1, 4, 8, 14]. Because at present there is no a consensus form of view model for object-oriented

[3]Subjects are the active entities and objects are the passive entities.

databases, we consider the view model which was proposed by Bertino [4] and describe how security properties might be incorporated into such a model. The view model, in the following, we describe is partly inspired from the one proposed in [4] for a context where security problems are not relevant.

In [4], Bertino provides a view model for object-oriented databases that extends in various directions typical view models of relational databases. Figure 4 in Appendix C illustrates the use of the view definition. As shown in [4], the view model also support schema changes such as changes to a class content, and changes to the class inheritance hierarchy. For a detailed description of keywords and discussion of how schema changes are supported by the view model, the reader is directed to [4].

Smith in [25] identified three "dimensions" of protection:

1. *the data itself may be classified;*
2. *the existence of the data may be classified; and*
3. *the reason, or rule, for classifying the data may be classified.*

Dealing with the first two dimensions of protection, in the following, we first introduce the notation *access-view.* Then because third security dimension is to classify rules which are used to classify data, we introduce the notation *security-constraint.* The set of access views and security constraints construct meta-classes ACCESS-VIEWS and SECURITY-CONSTRAINTS, respectively. The ACCESS-VIEWS are used for enforcement of mandatory access control, and the SECURITY-CONSTRAINTS, which contains rules that the view must satisfy, are used for controlling the security levels of views and their related views.

An instance of the ACCESS-VIEWS has the following format:

access-view av-name [parameters]
[**properties** list_of_property_names_and_security_levels]
[view-query]
[**additional-properties** list_of_properties_and_security_levels]
[**methods** list_of_method_specifications]
[**superviews** list_of_views]
[**OID** true_or_false]
[**identity-from** list_of_class_names]
[**av_level** view_security_level]
[**av_range** upper_and_lower_security_level]
[**mac-constraint set-of** <SECURITY-CONSTRAINTS> mac_names]

The *av-name* is the name of the access view, and must be different from the names of other access views, security constraints, and classes. Parameters are bounded to the actual values at the time the view is evaluated. So when the value of the parameters are determined, corresponding objects are evaluated dynamically.

As stated in [4], a view is defined by a query (view-query), as in relational databases, on an arbitrary sets of classes and derived classes (or views), called *base class (es).* Associated with each view is a set of instance variables and their security levels that may be derived from base classes (**properties**) or may be introduced (**additional-properties**). **OID**, which may be true or false, indicates whether persistent identifiers must be provided for the view instances or not (the default value is true); this is important for performance and avoids view instances to be re-evaluated each time on the whole database. If we want that views are evaluated dynamically, the **OID** must be false. The keyword **identity-from** is used for views, whose view-queries contain joins, to define a subset of the base classes on which the view instances depend for identity. A view also may have methods associated with it that they might be received from the base class or be renamed, or they are defined (**methods**).

Two types of view hierarchies may be formed. One is the *inheritance (or is-a) hierarchy* where a view has implicit or explicit super-views associated with it (**superviews**). The view inherits all of the instance variables, methods, and view constraints associated with its superclass (or super-view). For detailed discussion how inferred hierarchy for views is possible, see [2, 1].

The second view hierarchy is the *composite (or is-part-of) hierarchy.* Since an instance variable specification provided with the keyword **additional-properties** consists of the variable name and the domain where the domain can be a class or a view. Then an instance of a view may be the aggregation of a set objects, each of which belongs to some class or view. Such an aggregate view is sometime called a *composite object,* or here is called a *composite view.*

av_range indicates the range of the lowest security level and highest security level of view instances . **av_level** indicates the security level of the view. If the range is not indicated, then the view will be considered as a *single-level view;* all its properties have the same security level indicated by **av_level**.

mac-constraint provides the set of security rules that the view must satisfy. Each entry of that is an instance of the class SECURITY-CONSTRAINTS.

Employing the above view model, one possible implementation of foregoing examples in Section 1 is as follows. In the following examples, a portion of a university database shown in Figure 1 in Appendix A is used.

Example 1 *GAP instance variable of class STUDENT is SECRET.*

Solution: Create two following views. The level of *U_Student* is UNCLASSIFIED and the level of *S_Student* is SECRET.

access-view U_Student
select S.idno,S.name, S.age, S.status, S.address, S.spouse, S.sex, S.subject, S.Start-Date, S.Graduate-Date, S.takes
from S:STUDENT
av_level UNCLASSIFIED;

access-view S_Student
properties [Greatest_Point, SECRET]
select S.GAP
from S:STUDENT
superview U_Student
av_level SECRET
av_range [UNCLASSIFIED, SECRET]

The access view *U_Student* is a single-level view, and contains all variable instances of the base class STUDENT except GAP. It is labeled UNCLASSIFIED. The access view *S_Student* which is defined on top of *U_Student* is a multi-level view because in addition to secret instance variable *Greatest_Point* it inherits all properties of superview *U_Student* which are unclassified.

Example 2 *The address instance variable of EMPLOYEE is SECRET if the value of the instance variable, occupation, is a chancellor or a vice-chancellor.*

One possible solution is to create two following views. *U_Employee* and *S_Employee* are labeled UNCLASSIFIED and SECRET, respectively.

access-view U_Employee
select E
from E: EMPLOYEE
where not (E.occupation ="chancellor" **or** E.occupation ="vice-chancellor")
av_level UNCLASSIFIED;

access-view S_Employee
select E
from E: EMPLOYEE
where (E.occupation ="chancellor" **or** E.occupation ="vice-chancellor")
av_level SECRET;

As observed in the above example, without being necessary to redesign the schema, the enforcement of security constraint has been done.

There is a problem with above examples. Because the security level of the view *U_Employee* is unclassified, then unclassified users will know that there are expected employees classified at the higher level. In order to hide the classification rule but still allow it to be enforced, and to simplify verification and assurance process, we introduce the meta-class SECURITY-CONSTRAINTS which each its instance has the following format.

security-constraint sc-name [parameters]
[predicates] [property-name.**level** = property_security_level]
[predicates] [view-name.**level** = view_security_level]
[**sc_level** sc_security_level]

The *sc-name* is the name of the security constraint, and must be different from the names of other security constraints, classes, and access views. Parameters are bounded to the actual values at the time the view is evaluated. So when the value of the parameters are determined, corresponding security rules that the view must satisfy, are evaluated for controlling the security levels of view and its properties.

sc_level indicates the security level of the security constraint. Note that the security level of properties and views can be indicated either in the access view definitions or in the security constraint definitions.

The types of constraints include *simple, content, and aggregate security constraints* [12].

A *simple constraint* classifies entire view property, or view, e.g., the view property *Greatest_point* in view *S_Student* is secret (i.e., all *Greatest_point* values will be secret). The view *S_Student* can also be defined as follows.

access-view S_Student
properties [Greatest_Point]
select S.GAP
from S:STUDENT
superview U_Student
mac-constraint set-of < SECURITY-CONSTRAINTS > mac_S_Student
av_range [UNCLASSIFIED, SECRET]

security-constraint Sc_S_Student
Greatest_Point.**level**= SECRET
S_Student.**av_level** SECRET

If the statement *S_Student.mac_S_Student* = **insert***(Sc_S_Student)* is executed, then the address of *Sc_S_Student* will be added to the set of *mac_S_Student* automatically.

A *content-based constraint* provides a means of classifying data at the object or property value level by using a predicate based on the values of some objects and/or properties. For example, the security-constraint *Sc_Employee* in the following classifies the view *Av_Employee* at either secret or unclassified depending on the value of the property *occupation.* Using the security constraint definition, Example 2 can be redefined as follows.

access-view Av_Employee
select E
from E: EMPLOYEE
mac-constraint set-of < SECURITY-CONSTRAINTS > mac_Employee;

If we execute the statement *Av_Employee.mac_Employee* = **insert***(Sc_Employee)*, then the address of *Sc_Employee* will be added to the set of *mac_Employee* automatically.

security-constraint Sc_Employee
if (E.occupation ="chancellor" **or** E.occupation ="vice-chancellor")
then Av_Employee.**level** = SECRET
else Av_Employee.**level** = UNCLASSIFIED

An *aggregate constraint* classifies a collection of property values (say ten or more/less) or relationship among data at a higher security level. For example, if we define a view which retrieves accessories supplied by the specific supplier, the associated security constraint can be declared such that if the number of the accessories is more than ten, the view is classified at a higher level say SECRET.

4 Secure Multi-level View Model

Before we proceed to discuss the essential security properties for the model, it is important to stress two things. First, Our design supports multi-level objects at the view layer only. The multi-level views will be mapped onto single-level objects. Second, our design for the enforcement of mandatory security properties relies on an underlying security kernel.

As stated in [24], an object-oriented database is considered secure if:

1. No subject is able to obtain information without authorization.
2. No subject is able to modify information without authorization.
3. No mechanism exists whereby a subject authorized to obtain information can communicate that information to a subject not authorized to obtain it.
4. No subject is able to activate a method without authorization.

Issues (1), (2), and (4) are usually addressed by models to enforce discretionary access control, while issues (1) and (3) are normally addressed by models to enforce mandatory security. Therefore since our concern in this paper is to enforce mandatory security, we here present the security properties of a secure multilevel view object model that must be satisfied. How the discretionary access control can be enforced by a view object model, has been discussed in [2].

In the following, we first discuss the security properties associated with views. We then present the rules which are needed to obtain a view from a real single-level database. Finally, we consider the update problem and express some rules for that.

In our following discussion, we assume that each valid object in the database except atomic object and each valid user has a *security level* which is defined as a pair *(sensitivity level, category).* The *sensitivity levels* are taken from the list that is partially ordered by the ordering relation "$\geq$". The *category* is defined as a set (possible empty) of *compartments.* Compartments generally have no ordering but if a compartment is subdivided into other compartments, they are then ordered by set inclusion. Moreover, for security levels $X = (L1, C1)$ and $Y = (L2, C2)$, we define X *dominates* Y (denoted by $X \geq Y$) if and only if $L1 \geq L2$ and set $C2$ is subset of $C1$ ($C2 \subseteq C1$), and we define X *strict dominates* Y (denoted by $X > Y$) if and only if $L1 > L2$ and set $C2$ is a subset of $C1$ ($C2 \subseteq C1$).

We also assume that the following methods associated with meta-classes ACCESS-VIEWS and SECURITY-CONSTRAINTS are available to retrieve security information associated with an entity e. The entity e might be an access view, a security constraint, an attribute, a method, a class, or an object.

Level(e)	Display the security level of entity e. Entity e can be an access view, or a security constraint.
Lower(e)	Display the lower level range of entity e.
Upper(e)	Display the upper level range of entity e.

Denote by LUB the least upper bound, and GLB the greatest lower bound.

4.1 View

A *view name* is the external representation of an access view. Then when a user wants access a view definition, (s)he must first authorized to access *view name.* Every access view v is defined by a view specification which has a security level, **av_level**. Hence in order that a user is able to access the view, the security level of the view must be dominated by the security level of the user.

Rule 1 *(Subject Property). A user u can access an access view v if the security level of the user u dominates the security level of the access view v in case that the view v is single-level view (v.**av_level** $\leq$ **Level**(u)), or if the view v is a multi-level view, the lowest security range of the view v is dominated by the security level of the user u (***Lower***(v) $\leq$ **Level**(u)).*

The Rule 1 is the simple property specified in the Bell-LaPadula security policy. Only *read-up* is permitted. Users with security level lower than the security level of a view are not able to access the view definition and consequently the view instances. Moreover in order that the user u can access properties and methods associated with the view v, which since now we refere them as view facets, the following rule must be held.

Rule 2 *The user u is able to access facet x of the view v if its security level is dominated by the security level of the user u.*

$$\textbf{Level}(v.x) \leq \textbf{Level}(u)$$

In general, a view can be constructed in two distinct ways [2, 1]: *top-down* or *bottom-up.* In the *top-down* approach, large classes (or views) 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 (or views) 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). Therefore because in most cases a view may contain much information, it must be classified at the highest level.

Rule 3 *(View Property). If an access view v is constructed on classes (or views) $v_1, v_2, \ldots, v_n$, the security level of v must be satisfied following*

$$v.\textbf{av_level} \geq LUB\{v_1.\textbf{av_level}, v_2.\textbf{av_level}, \ldots, v_n.\textbf{av_level}\},$$

and the view range must contain the security level of v

$$\textbf{Lower}(v) \leq v.\textbf{av_level} \leq \textbf{Upper}(v).$$

Furthermore since when a view is populated, it contains some database objects. Then the security level of the view must also constraint by the following rule.

Rule 4 *(View-instance Property). If the population of a view v is objects $o_1, o_2, \ldots, o_n$, then the security level of the view v must satisfied:*

$$v.\textbf{av_level} \geq LUB\{\textbf{Level}(o_1), \textbf{Level}(o_2), \ldots, \textbf{Level}(o_n)\},$$

and the range of the view v must contain the greatest lower bound (GLB) and the greatest upper bound (LUB) of the security level associated with objects.

The set of database objects contained in the view population is controlled by the view specification and/or a set of security constrains associated with it. Hence, based on the information contained in constraints, they must also be classified. Every *security-constraint* is defined by a set of constraints, and a security level (**sc_level**).

Rule 5 *The security level of a security-constraint is the least upper bound of the security levels of all facets (properties and methods) and all literal security levels contained in the constraints, ie. if* $x_1, x_2, \ldots, x_n$, *and* $s_1, s_2, \ldots, s_m$ *are facts and nominated security levels contained in the security-constraint sc, respectively, then the security level of the sc must satisfied:*

$$sc.\mathbf{sc_level} \geq LUB\{\mathbf{Level}(x_1), \mathbf{Level}(x_2), \ldots, \mathbf{Level}(x_n), s_1, s_2, \ldots, s_m\}.$$

The above rule indicates that the security level of a security constraint must be at least equal to the least upper bound of security levels of information contained by constraints. For example, by applying the above rule on *Sc_Employee* in Example 2 of Section 2, the security level of the Sc_Employee must be at least SECRET. If the **sc_level** is not indicated by the the user, the computed security level will be the default value of **sc_level**.

Rule 6 *A user u can access a security-constraint sc if the security level of the sc is dominated by the security level of the user u,* $\mathbf{Level(u)} \geq sc.\mathbf{sc_level}$.

So the security-constraint *Sc_Employee* is only accessible to users with clearance secret or higher.

Properties and methods (facets) of an access view can be derived from base views or classes, or can be defined additionally. Moreover, the security level associated with each facet can also be derived or defined. The Rule 7 and 8 describes the security properties associated with the *view facets.*

Rule 7 *(View Facet Property). If x is a facet (a property or a method) of a view v and is derived from a base view w, the security level of x must be satisfied:*

$$\mathbf{Level}(v.x) \geq v.\mathbf{av_level},$$

and

$$\mathbf{Level}(v.x) \geq \mathbf{Level}(w.x).$$

If x is defined or redefined in v, the security level of x must dominate the security level of v, ie.,

$$\mathbf{Level}(v.x) \geq v.\mathbf{av_level}.$$

Rule 8 *The security range of a view must contain the security level of all facets contained in the view, ie., if* $x_1, x_2, \ldots, x_n$ *are facets of v, then the following must be held:*

$$GLB\{\mathbf{Level}(v.x_1), \mathbf{Level}(v.x_2), \ldots, \mathbf{Level}(v.x_n)\} \geq \mathbf{Lower}(v),$$

and

$$LUB\{\mathbf{Level}(v.x_1), \mathbf{Level}(v.x_2), \ldots, \mathbf{Level}(v.x_n)\} \leq \mathbf{Upper}(v).$$

In [2], three types of inheritance among views are distinguished, *strict constraint inheritance, constraint inheritance,* and *specialization inheritance.*

In *constraint inheritance,* views consist of all object instances of base classes (or views) that satisfy given constraints.

In *strict constraint inheritance,* not only the set of database objects contained by a view is constrained, but the view's properties are fewer than the base classes (or views).

In *specialization inheritance,* not only views inherit all the properties of base classes (or views), but they also have some additional properties and/or methods.

In the first two cases, the amount of the information that a sub-view has, is less than the information contained by the base view, so the security level of the sub-view may be as much as the security level of the base view.

Rule 9 *(Hierarchy Property). If view v inherits the view w, depending on the type of the inheritance, one of the following must be held.*

If the relationship is strict constraint or constraint inheritance, then the security level of the v must be equal to the security level of w.

$$\textbf{Level}(v) = \textbf{Level}(w)$$

If the relationship is specialization inheritance, the security level of the v must dominate the security level of w.

$$\textbf{Level}(v) \geq \textbf{Level}(w)$$

In both cases, the set of the constraints of the view v must contain the set of the constraints of the view w.

A view may inherit or derive the facets from one or more super-views or base-views. In the case of conflict, the following rule will be used to resolve the conflict.

Rule 10 *(Multiple Inheritance). If a view v inherits the facet x from views $v_1, v_2, \ldots, v_n$, then the security level of x in v must dominate the least upper bound of the security levels of x in v_i $(1 \leq i \leq n)$.*

$$\textbf{Level}(v.x) \geq LUB\{\textbf{Level}(v_1.x), \textbf{Level}(v_2.x), \ldots, \textbf{Level}(v_n.x)\}$$

If there are more than one such v_j, then some priori rule must be enforced to resolve the conflict.

As stated before, it is possible to define composite (or aggregation) hierarchy on views; the domain of the instance variable provided with the keyword **additional-properties** for a view property can be a view or a class. Then a view v may be a composite of views $v_1, v_2, \ldots, v_n$. The security level of v is constrained by the following rule:

Rule 11 *(Composite Property). If a view v is composite of the views (or classes) $v_1, v_2, \ldots, v_n$, then the security level of the v must satisfy:*

$$v.\textbf{av_level} \geq LUB\{v_1.\textbf{av_level}, v_2.\textbf{av_level}, \ldots, v_n.\textbf{av_level}\}.$$

4.2 Derivation Rules

A derivation rule specifies how derived objects are computed. We therefore here describe the rules which are needed to obtain a view from a real single-level database.

Because views are also classified, as a user is cleared to access a view (the user's security level dominates the security level of the view), the view objects are evaluated as follows:

Rule 12 *(Single-level View Instantiation). If an object in the database has a security level dominated by the security level of a view, the object may be derived as an instance of the view.*

Note that a view may be evaluated dynamically [4]. Then view instances are created only if a user or process wants to perform it.

Rule 13 *(Multi-level View Instantiation). If the security level of the view property dominates the security level of the corresponding property value of an object in the database, then that object may be derived an instance of the view.*

All derived single-level objects corresponding to the view properties are joined to instantiate a multi-level view. As said through Rule 4, the security level of derived data is at least the least upper bound of the security levels associated with the derived data. Those derived data will be presented to a user which their security levels are dominated by the security level of the user. Next section presents how derived data can be stored back into the database.

4.3 Update and Object Creation

Data entering the database as inserts or updates may be performed by any views, but the following rules must be enforced.

Rule 14 *(Insertion). The security level of the inserted object according to a set of security constraints associated with views will be obtained. If the computed security levels for the entire object is unique, then the single security level is assigned to the object and the single-level object is stored into the database. But if the computed security levels are different for every properties, the entering datum is decomposed to single-level objects according the computed security levels, and they are then stored.*

According to the above rule, if the inserted object is multi-level, since the underlying database is single-level, the multi-level object must be decomposed to several single-level objects. Solution to the object decomposition was proposed in [5].

Rule 15 *(Updating). For every view property value, the value with the computed security level is stored back into the database if: 1) the computed security level dominates the security level associated with the user on whose behalf it was computed, and the computed security level is equal to the security level of the corresponding property in the database; or 2) the downgrade is authorized and confirmed by the security officer.*

5 Aggregation and Inference

5.1 Aggregation

The aggregation problem occurs when a user can form aggregates of related data items, all of which are classified at some level, that deduces classified data. Here, it is the collecting together of some amount basic lowly classified data which is highly classified.

Example 3 *Personal specification and monthly wage of academic staffs are unclassified, but the association of a particular monthly wage with a specific individual academic staff is classified secret.*

One possible solution is that two unclassified views for personal specification and monthly wage of the academic-staff, *U_Academic_Staff and U_Monthly_Wage,* are declared respectively. The secret class *S_Academic_Monthly_Wage* is declared to protect the association academic specifications and their monthly wages.

It is worth noting that by declaring above views, the protection against inference is not necessarily complete. For example, if there were only a few academic staffs and wages, and the rank of the academic staffs were known or could be guessed, the association could be reconstructed. In such cases, the preferred solution would be to classify one of the two entities in the relationship at the higher level. For example, putting the monthly wages in a separate view at the secret level would protect the association, but it would also, in effect, classify the monthly wages.

In general, the aggregation problem can be stated in terms as follows: If a view v is the aggregate of views $v_1, v_2, \ldots, v_n$ and the security level of v strictly dominates the security level of all v_i $(1 \leq i \leq n)$, then the aggregation problem occurs.

$$v.\textbf{av_level} > v_i.\textbf{av_level}$$

for all i, $1 \leq i \leq n$. To solve that, the following rule must be held:

Rule 16 *If a view v is the aggregate of views $v_1, v_2, \ldots, v_n$, then it should exist at least a v_i such that it security level is equal to the security level of the v.*

$$v.\textbf{av_level} = v_i.\textbf{av_level}$$

for some i, $1 \leq i \leq n$.

5.2 Inference

Inference problem occurs when a user can deduce (or infer) information from a collection of individual accesses against a database. Solutions to the inference problem were proposed in [11, 22, 17, 15, 27, 7] among others in the context of statistical and relational databases. Several approaches are used to handle the problem: (1) place restrictions on the set of allowable queries that can be issued by a user; (2) add noise to the data; and (3) augment a database with a logic-based inference engine to modify queries before the database processes them.

Since views, if properly defined and controlled, can be regarded as a constraint relating derived data to other data that are stored or derived; and can be used to constraint the user to only the data that he actually needs to see, they provided exactly the right basis for addressing the inference problem or at least decreasing the *size* of the inference problem. It is true firstly because they represent the real-world semantic relationships among data, and it is those relationships that are exploited during inference; and secondly since constraining a user to a smaller portion of the database through the proper use of the views than (s)he would have access to the database without the use of views, will reduce the number and types of queries that (s)he can write.

It is worth noting that some researchers believes that aggregation and inference are the different faces of the same problem [19].

6 Polyinstantiation

Polyinstantiantion generally occurs when two subjects at different security levels see two different forms of a single entity in the real world. In a multi-level view model, the different forms of an entity could relate to different view definitions, to same view-view-name! identified different views with different security-levels, and to different entity values. So we have three possible polyinstantiations.

- *View-name polyinstantiation.* An actual view is identified by the pair: a view-name v and the security-level (**av-level**). So that there may be several views identified by the same view-name v which correspond to different security-levels.

- *View-definition polyinstantiation.* A view may contain different facets with different security-levels. For instance, an unclassified user sees a view v which contains the properties *(idno, name)* and has a method *change-name*. A secret user sees v which contains *(idno, name, salary)*, and has two methods *change-salary* and *change-name*.

- *View-instance polyinstantiation.* An object is identified by an object identifier (OID) and an associated security-level, so that a multi-level view may contain several object instances for an OID corresponding to different security-levels.

A polyinstantiated view-name may arise whenever an unclassified (or confidential) user requests to use the same name for defining a view which is already used by a secret user. To handle this type of polyinstantiation, we suggest that a view name associated with the security-level of the user is used to name the view. For instance, for a view v, the unclassified user will use *U-v* and the secret user will use *S-v*.

A polyinstantiated view-definition may occur whenever an unclassified user modifies the view definition and includes the same facets which already exist at higher security level. This type of polyinstantiation will not occur in our model because the security level of a view must be the least upper bound of all security levels of all facets contained in the view (Rule 3). The view can be accessed by a user u if the security level of u dominates the security level of the view (Rule 1).

A polyinstantiated view-instance may happen if two subjects at different security level request the same identifier for two different objects which represent two different entities. Therefore, if globally unique object identifiers are used to identify objects, the polyinstantiation will not occur in the model.

Boulahia-Cuppen et. al. in [5] presented a decomposition algorithm for OODBS which provides the possibility of choosing globally unique object identifiers (by partitioning the set of object identifier into several pairwise disjoint subsets associated with each security level). For instance a multi-level object O, which is an instance of a multi-level view *S_Student,* will be decomposed to *U-O, C-O,* and *S-O*. Each of them is actually a single-level object corresponding to unclassified, confidential, and secret which is physically stored in a single-level database. Dynamic links are also created between these objects. For example, in the *S-O,* there are pointers to the confidential properties stored in the *C-O.* In the *C-O,* there are pointers to unclassified properties of the *U-O*. This means that if an classified user updates the unclassified properties of *U-O,* the update will be automatically propagated to the instances of *C-O* and *S-O.* Note that the values of the properties which pointer to low-level objects can be updated by the users cleared to access them. If this happens, the pointer to the low-level database is broken, and the value of the object in the low-level database stands as a cover story. For detailed discussion, the reader is directed to [5].

Therefore by employing the above approach to decompose multi-level views into single-level objects and selecting object identifiers corresponding to each security levels, view-instance polyinstantiation does not occur in our model. Furthermore, there is no *referential ambiguity* as all references to an object use the unique OID.

Note that although there is no view-instance polyinstantiation because OIDs are unique, but it is possible that the value of the property *idno* is not unique. One way of avoiding this is the use of security constraints that requires all instances of *PERSON* to have the same security level.

7 Evaluation of the Proposed Model

Gajnak in [13] chooses a view of a multi-level secure database as a set of associated facts, and presents three general principles for a *well formed* multi-level secure database. The principles are: *the granularity principle, the dependency principle,* and *the determinacy principle.* The granularity principle states that the finest level of granularity for the protection purpose should be a structure which correspond to atomic facts. The dependency principle requires that the security level of a fact dominate the security level of any other fact it depends upon. The determinacy principle states that factual dependencies should not be ambiguous.

To evaluate the logical consistency of the proposed model, we assess the model with respect to these three fundamental principles for a multi-level secure database model. To do so, we must first determine the meaning of the *atomic facts* and *factual dependency* in the proposed model.

A *fact* is an encapsulated unit of information. Atomic facts are facts that do not depend on other facts. Six types of atomic facts can be distinguished in a multi-level view model. They are:

1. the fact of the object existence (which is presented by the object identifier),
2. the association between an object and the values of its properties,

3. the association between an object and a view-instance,

4. the association between a view definition and a view name,

5. the association between a view definition and a security constraint, and

6. a hierarchical association (inheritance or composite).

The following four factual dependencies also exist among these facts.

1. The fact of association of a property value with an object depends on the existence of the object.

2. The fact of association of a view-instance with object depends on the existence of objects.

3. The fact of association of a view-name with a view-definition depends on the existence of the view definition.

4. The fact of association of a security constraint with a view definition depends on the existence of the view definition.

Now, consider the three principles, *granularity, dependency,* and *determinacy.* The granularity principle states that the finest level of labeling granularity must cover all the above mentioned facts. This is exactly the entities that we assumed to be labeling.

The dependency principle states that the security level of an association must dominate the security level of its components. Rules 3-7, 4-5, and 9-11 take care of that.

The determinacy property addresses the problem of interpreting the database in the face of polyinstantiation. As discussed before because we assume that globally unique OIDs are used to identify objects, then the polyinstantiation will not occur in the model.

We believe that the proposed multi-level view model works because we have been able to map its constructs to the set of associated facts and because it directly supports the three principles of multi-level data: granularity, dependency, and determinacy.

8 Conclusions and Remarks

In this chapter, our objective was to provide a multi-level view model derived from a single-level secure OODBSystem. The model allows us to use the existing security kernel. It also overwhelms the difficulties of handling OODBS with multi-level objects. One distinct advantage of our approach is that the multi-level view model relies on an underlying security kernel for the enforcement of mandatory security properties.

Our model can be seen as an extension of the view model proposed in [4]. We introduced the notion of *security constraint* which is associated with each view. We also discussed the usage of the views and security constraints to handle simple, content-dependent, and context-dependent classifications. We have then described the multi-level security properties for a secure multi-level view model based on a secure single-level OODBSystem. Finally, we discussed issues such as aggregation, inference, and polyinstantiation problems.

Note that two types of users require access to views: the database *security officers* and *the end users.* It is clear that the security officers require unrestricted access to the views in order to define and maintain the database. End users require some access to the view definitions in order to be able to query the database. Two policy approaches have been proposed for this access: (1) to allow users to browse through all external schema, or (2) to attached discretionary access rights on the views. The first approach violates the least privilege principle. Then the second approach should be used. In other words, a discretionary access control on top of the multi-level view model should be implemented. In [2], we proposed a discretionary access control model using views.

References

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

[2] A. Baraani-Dastjerdi, J. Pieprzyk, R. Safavi-Naini, and J. R. Getta. A Model of Authorization for Object-Oriented Databases based on Object Views. In T. Ling, A. Mendelzon, and L. Vieille, editors, *Proceedings of The Fourth International Conference on Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 503–520, Singapore, December 1995. Springer-Verlag.

[3] D.E. Bell and L.J. LaPadula. Secure Computer System: Unified Exposition and Multic Interpretation Report ESD-TR-75-306. Technical report, MITRE Corp., Bedford, Mass., March 1976.

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

[5] N. Boulahia-Cuppens, F. Cuppens, A. Gabillon, and K. Yazdanian. Decomposition of Multilevel Objects in an Object-Oriented Database. In *Computer Security ESORICS 94, Third European Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 375–402. Springer-Verlag, November 1994.

[6] B. G. Claybrook. Using Views in A Multilevel Secure Database Management System. In *Proceedings of the 1983 IEEE symposium on security and privacy*, pages 4–17. IEEE Computer Society Press, April 1983.

[7] F. Cuppens. A Model Logic Framework to Solve Aggregation Problems. In C. E. Landwehr and S. Jajodia, editors, *Database Security V*, pages 315–333. Elsevier Science Publishers B. V. (North-Holland) IFIP, 1992.

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

[9] D. E. Denning, S. G. Akl, M. Heckman, T. F. Lunt, M. Morgenstern, P. G. Neumann, and R. R. Schell. Views for Multilevel Database Security. *IEEE Transactions on Software Engineering*, SE-13(2):129–140, February 1987.

[10] D. E. Denning and T. F. Lunt. A Multilevel Relational Data Model. In *Proceedings of Symposium on Computer Security and Privacy*, pages 220–234, Oakland, CA., 1987. IEEE Computer Society Press.

[11] D. E. Denning and J. Schlorer. Inference Control for Statistical Databases. *Computer*, 16:69–82, July 1983.

[12] P. A. Dwyer, G. D. Jelatis, and M. B. Thuraisingham. Multilevel Security in Database Management Systems. *Computers & Security*, 6(3):252–260, June 1987.

[13] G. E. Gajnak. Some Result from the Entity/Relationship Multilevel Secure DBMS Project. In T.F. Lunt, editor, *Discussions of topics presented at a Workshop held at the Vallombrosa, Conference and Retreat Centre, Menlo Park, CA May 1988, Research Directions in Database Security*, pages 173–190. Springer-Verlag, 1992.

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

[15] T. H. Hinke and H. S. Delugach. AERIE: An Inference Modeling and Detection Approach For Databases. In B. M. Thuraisingham and Landwehr, editors, *Database Security VI*, pages 179–193. Elsevier Science Publishers B. V. (North-Holland) IFIP, 1993.

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

[17] T. F. Keefe, M. B. Thuraisingham, and W. T. Tsai. Secure Query Processing Strategies. *IEEE Computer Society Press*, 22(3):63–70, March 1989.

[18] T. F. Keefe and W. T. Tsai. Prototyping the SODA Security Model. In D. L. Spooner and C. E. Landwehr, editors, *Database Security II*, pages 211–235. Elsevier Science Publishers B. V. (North-Holland) IFIP, 1990.

[19] T. Y. Lin. Inference Secure Multilevel Databases. In B. M. Thuraisingham and Landwehr, editors, *Database Security VI*, pages 317–333. Elsevier Science Publishers B. V. (North-Holland) IFIP, 1993.

[20] T. F. Lunt. Multilevel Security for Object-Oriented Database Systems. In D. L. Spooner and Landwehr, editors, *Database Security III*, pages 199–209. Elsevier Science Publishers B. V. (North-Holland) IFIP, 1990.

[21] T. F. Lunt. SeaView. In T. F. Lunt, editor, *Discussions of topics presented at a Workshop held at the Vallombrosa, Conference and Retreat Centre, Menlo Park, CA May 1988, Research Directions in Database Security*, pages 2–32. Springer-Verlag, 1992.

[22] N. S. Matloff. Another look at the use of noise addition for database security. In *Proceedings of the 1986 Symposium on Research in Security and Privacy*, pages 173–180, Oakland, CA., 1986. IEEE Computer Society Press.

[23] 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*, pages 260–272, Oakland, CA., May 1992. IEEE Computer Society Press.

[24] 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.

[25] G. W. Smith. Identifying and Representing the Security Semantics of an Application. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference.* , December 1988.

[26] P. D. Stachour and B. Thuraisingham. Design of LDV: A Multilevel Secure Relational Database Management System. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):190–209, June 1990.

[27] M. B. Thuraisingham. Security Checking in Relational Database Management System Augmented with Inference Engines. *Computers & Security*, 6(6), December 1987.

[28] 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.

[29] J. Wilson. Views as the Security Objects in a Multilevel Secure Relational Database Management System. In *Proceedings of Symposium on Computer Security and Privacy*, Oakland, CA., April 1988. IEEE Computer Society Press.

A Appendix

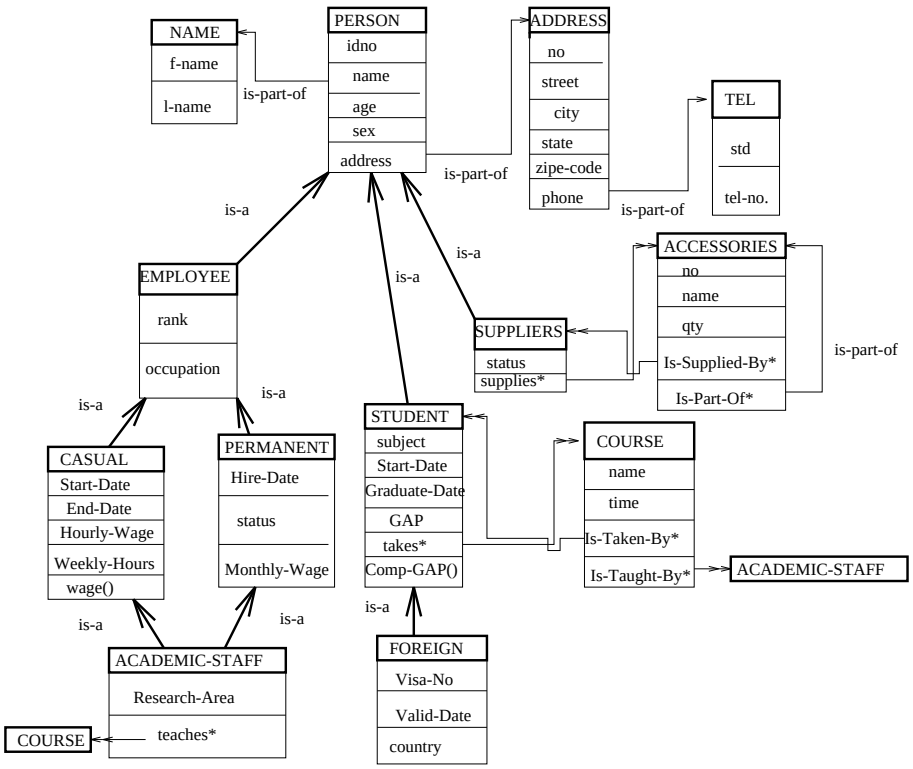

Figure 1: A portion of University Database.

B Appendix

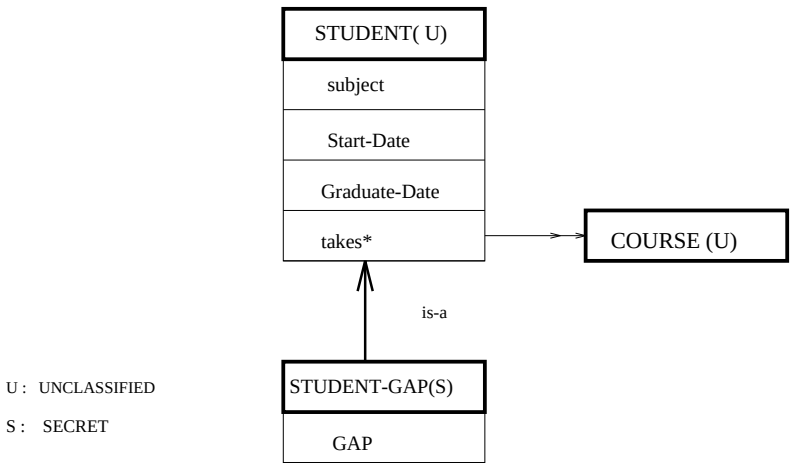

Figure 2: Simple Constraint.

C Appendix

This view defines a view called *Wage* consisting of the employee names and salaries for all employees with salaries greater than X where the actual values of X is determined at the evaluation time of the view. This is termed a *parameterized view* (for full description see [2]).

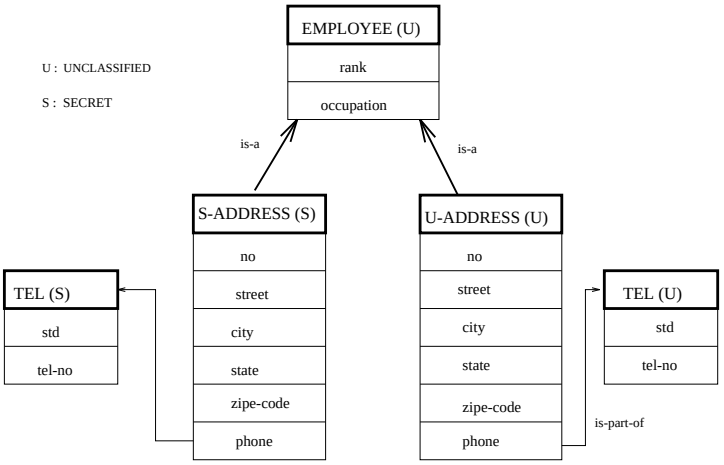

Figure 3: Content Constraint.

Example 4 *Define a view that lists all employees whose monthly salary are more than X.*

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

The method *salary* computes the monthly earning for casual employees. 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

Figure 4: Sample example 1.