A Model of Authorization for Next-Generation Database Systems

FAUSTO RABITTI, ELISA BERTINO, WON KIM, and DARRELL WOELK
Microelectronics and Computer Technology Corporation

The conventional models of authorization have been designed for database systems supporting the hierarchical, network, and relational models of data. However, these models are not adequate for next-generation database systems that support richer data models that include object-oriented concepts and semantic data modeling concepts. Rabitti, Woelk, and Kim [14] presented a preliminary model of authorization for use as the basis of an authorization mechanism in such database systems. In this paper we present a fuller model of authorization that fills a few major gaps that the conventional models of authorization cannot fill for next-generation database systems. We also further formalize the notion of implicit authorization and refine the application of the notion of implicit authorization to object-oriented and semantic modeling concepts. We also describe a user interface for using the model of authorization and consider key issues in implementing the authorization model.

Categories and Subject Descriptors: H.2.0 [Database Management]: General
General Terms: Security, Theory
Additional Key Words and Phrases: Object-oriented database, semantic database

1. INTRODUCTION

An authorization mechanism is an integral part of most commercial database systems. A model of authorization must be designed to be consistent with the data model supported in a database system. The models of authorization supported in existing database systems are all designed for relational, hierarchical, or network models of data. Rabitti, Woelk, and Kim point out a few major shortcomings in these models that make them inadequate for richer data models, such as object-oriented and semantic data models, and propose a model of authorization in an attempt to remove these shortcomings [14]. The existing models assume that the unit of authorization is a relation (record type) or an attribute of a relation (record field). Furthermore, they do not...
reflect the semantics of object-oriented concepts such as the class hierarchy and methods, and semantic modeling concepts such as composite objects (part hierarchy of objects) and versions.

The model of [14] refines the notion of implicit authorization first formalized in [5] and [6], and extends it to object-oriented and semantic modeling concepts. The idea behind implicit authorization is that an authorization of a certain type defined for a user on a certain database object implies other authorizations. An authorization is specified in terms of a subject (a user, a group of users), an authorization type (read, update, create), and an authorization object (a single object, a group of objects, an entire database). An explicitly specified authorization may imply authorizations along any combination of the three dimensions in authorization definitions, namely, the subject, authorization type, and authorization object. For example, an update authorization for a group of users on a group of objects (a class, a relation, a record type) implies the following authorizations: an update for any member of the group of users on any member of the group of objects, and a read for any member of the group of users on the group of objects or its members. The concept of implicit authorizations makes it unnecessary to store all authorizations explicitly; the authorization mechanism can compute authorizations from a minimal set of explicitly stored authorizations in screening any attempt to penetrate the system. Of course, the compute overhead inherent in implicit authorizations must be weighed against the savings in storage space in particular application environments.

In refining the concept of implicit authorizations, Rabitti, Woelk, and Kim also refined the specification of authorization [14]. An authorization type is distinguished as a strong or weak authorization, depending on whether any of the authorizations implied by the authorization may be overridden. Furthermore, to allow exceptions to an authorization, the model distinguishes an authorization as a positive or negative authorization. A negative authorization is an explicit denial of an authorization. In existing models of authorization, the absence of an authorization is used in place of a negative authorization; however, the absence of an authorization cannot be used to override another authorization.

Although [14] contains significant new results in the area of database authorization, the results are incomplete. The model suffers from a few important shortcomings. First, the model is based on the assumption that there is a total ordering of authorization types. If only the read and update authorization types are considered, the assumption is valid; however, as soon as we add additional authorization types, such as the generate (create) authorization type, the assumption breaks down. Second, the model uses excessively many types of authorization objects, making the model cumbersome. This problem manifests largely in the application of the concept of implicit authorizations to object-oriented and semantic modeling concepts. Third, although the model includes a clean formalization of strong authorizations, it does not formalize the semantics of weak authorizations, that is, authorizations implied by a weak authorization and the impacts of overriding such authorizations.
In this paper we present a model of authorization for next-generation database systems that support rich data models that incorporate object-oriented concepts and a number of important semantic modeling concepts, such as composite objects and versions. This model may be viewed as a significant extension of the authorization model developed for relational database systems, such as that described in [9]. It is based on the model of [14], but removes all of the major shortcomings mentioned above. Furthermore, we make the discussion of the model concrete by including a description of a user interface for invoking the capabilities of the model. We also consider key issues in implementing the model. We note, however, that considerations of such issues as mandatory security and authorizations based on the contents and contexts of the database, although important, are outside the scope of this paper. The interested reader may find preliminary discussions of these issues in [15].

The remainder of this paper is organized as follows: In Section 2 we present the basic concepts constituting our model of authorization, including the notions of implicit authorization, positive/negative authorization, and strong/weak authorization. Section 3 develops rules for computing implicit authorizations along each of the three dimensions in authorization definitions, namely, authorization subjects, authorization types, and authorization objects. In Section 4 we extend the authorization model to object-oriented and semantic modeling concepts, including the properties of a class, class hierarchy, composite objects, and versions. In Section 5 we consider implementation issues for the authorization model. Section 6 summarizes the paper.

2. BASIC CONCEPTS

In this section we begin with a brief review of object-oriented concepts found in most object-oriented programming and database systems. Then we present and extend the basic concepts of the authorization model originally developed in [14]; we start with an intuitive overview of the concepts and follow it up with a formalization.

2.1 Review of Object-Oriented Concepts

In object-oriented systems and languages, any entity is uniformly modeled as an object. Furthermore, an object is associated with a unique identifier. Every object encapsulates a state and a behavior. The state of an object is the values for the attributes of the object, and the behavior of an object is the set of methods that operate on the state of the object. The value of an attribute of an object is also an object in its own right. Furthermore, an attribute of an object may take on a single value or a set of values.

All of the objects that share the same set of attributes and methods are grouped into a higher level object called a class. An object must belong to only one class as an instance of that class. A class may be a primitive class, such as integer, string, and Boolean; a primitive class is a class with no attributes.
The value of an attribute of an object, since it is necessarily an object, also belongs to some class. This class is called the domain of the attribute of the object. The domain of an attribute may be any class, including a primitive class. The fact that the domain of an attribute may be an arbitrary class gives rise to the nested structure of the definition of a class. That is, a class consists of a set of attributes (and methods), the domains of some or all of the attributes may be classes with their own sets of attributes, and so on. Then the definition of a class results in a directed graph of classes rooted at that class; we call this graph a class-composition hierarchy. If the graph for the definition of a class is restricted to a strict hierarchy, the class becomes a nested relation.

Object-oriented systems allow the user to derive a new class from an existing class; the new class, called a subclass of the existing class, inherits all of the attributes and methods of the existing class, called the superclass of the new class, and the user may specify additional attributes and methods for the subclass. A class may have any number of subclasses. Some systems allow a class to have only one superclass, whereas others allow a class to have any number of superclasses. In the former, a class inherits attributes and methods from only one class; this is called single inheritance. In the latter, a class inherits attributes and methods from more than one superclass; this is called multiple inheritance. In a system that supports single inheritance, the classes form a hierarchy, called a class hierarchy. If a system supports multiple inheritance, the classes form a rooted directed acyclic graph, sometimes called a class lattice.

The concept of a class hierarchy is completely orthogonal to that of a class-composition hierarchy. A class hierarchy captures the generalization relationship between one class and a set of classes specialized from it. A class-composition hierarchy has nothing to do with inheritance of attributes and methods.

2.2 Intuitive Overview of the Basic Authorization Concepts

Basic Definition. An authorization is defined in [7], [8], [13], and [14] as a 3-tuple \((s, o, a)\), where

\[
\begin{align*}
S & \subseteq S, \\
o & \subseteq O, \\
a & \subseteq A,
\end{align*}
\]

the set of subjects (i.e., users) in a system;

\[
o \subseteq O,
\]

the set of authorization objects in a system; and

\[
a \subseteq A,
\]

the set of authorization types.

A function \(f\) is defined to determine if an authorization \((s, o, a)\) is True or False:

\[
f: S \times O \times A \rightarrow \{\text{True}, \text{False}\}.
\]

Given a triplet \((s, o, a)\), if \(f(s, o, a) = \text{True}\), then subject \(s\) has an authorization of type \(a\) on object \(o\).
Implicit Authorization. The easiest (but very storage-inefficient) way to implement the function \( f \) is to store explicitly all the triplets \((s, o, a)\) for which \( f \) is True. However, a value of \( f \) may imply other values along any of the \( S \), \( O \), and \( A \) domains of \( f \). For example, a user who has a read authorization on a class should be able to read all instances of the class: Authorizations are implied along domain \( O \). A manager should be able to access any information that his employees may access: Authorizations are implied along domain \( S \). A user with a write authorization on an object should be able to read that object: Authorizations are implied along domain \( A \).

Figure 1 illustrates a database granularity hierarchy. This hierarchy defines how objects in a system are organized in terms of other objects (this is not related to the class hierarchy or class-composition hierarchy). Given an object \( o \), the hierarchy defines how \( o \) is composed of objects at successively lower levels. For example, the object database [inventory] is composed of the classes Vehicle, Automobile, 4-Wheel-Vehicle, and Motor-Vehicle. Figure 2 illustrates one implicit authorization and the authorizations it implies.

Therefore, rules for deducing the value of \( f(s_1, o_1, a_1) \) from some explicitly defined \( f(s_2, o_2, a_2) \) obviate the need to store \( f \) for all points in \( S \times O \times A \) and also allow detection of conflicts between values of \( f \). The formal model of authorization described in this paper is essentially a set of such rules.

Implicit authorizations present a compute overhead to determine whether a particular authorization is implied by an explicitly stored authorization. This compute overhead must be carefully weighed against the space advantages that implicit authorizations offer. In general, if the need to define authorizations on individual objects (records) is not high, the case for implicit authorizations is correspondingly low. In relational databases, the storage advantage of implicit authorizations has been outweighed by the compute overhead. One of the reasons is that the unit of access in relational databases

---

ACM Transactions on Database Systems, Vol. 16, No 1, March 1991
is a relation, rather than any specific tuple of a relation; that is, access to a relational database is accomplished through a query formulated against a relation or a set of relations. Therefore, the case for defining authorizations on individual tuples, and consequently for implicit authorizations, is not very strong. For next-generation database systems that support the notions of objects, composite objects, and versioned objects, the need for implicit authorizations is significantly stronger. We discuss these concepts in greater detail later. However, briefly, a composite object is a potentially large set of objects related through the IS-PART-OF relationships, and a versioned object consists of a set of versions that are related through the IS-VERSION-OF relationships. It is useful to define a single authorization on a composite object or a versioned object and to have the same authorization apply implicitly to all constituents of the object. Furthermore, in object-oriented databases, an object is the logical unit of database access, since it is associated with a unique identifier. Therefore, it makes sense to treat an object as a unit of authorization.

The concept of implicit authorization is at the core of the model, and is further refined with the notions of positive and negative authorization, and strong and weak authorization, as follows:

**Positive/Negative Authorization.** The authorization mechanism in database systems generally assumes that a subject has a null authorization on every object until explicitly authorized [7]. The lack of an authorization on an object means that a subject cannot access the object. The concept of a
negative authorization complements the positive authorization. A subject $s$ may be denied access to an object $o$ because either $s$ has no authorization on $o$ or $s$ has a negative authorization on $o$.

**Strong/Weak Authorization.** An authorization in existing models is a strong authorization, in the sense that authorizations implied by some authorization cannot be overridden. A weak authorization allows exceptions in the implicit authorizations. A strong authorization guarantees that it and all of the authorizations it implies cannot be overridden, whereas authorizations implied by a weak authorization may be overridden by other (strong or weak) authorizations. Thus, in Figure 2 if a user $s$ has a strong write authorization on database[Inventory], $s$ also has implicit write authorizations on all objects in this database.

Figure 3 shows an example of a weak authorization with weak exceptions. In this example, $s$ is granted an explicit weak write authorization on database[Inventory]. Furthermore, $s$ is granted an explicit weak negative write authorization on the class Motor-Vehicle and an explicit weak negative read authorization on the class Vehicle. These two authorizations represent exceptions to the authorization on database[Inventory]. In Figure 2 the user receives implicit authorizations on all the objects belonging to the subtree rooted at database[Inventory]. In Figure 3 the resulting implicit weak autho-
A Model of Authorization for Next-Generation Database Systems

In Figure 4, implicit weak authorization with strong exceptions is illustrated. The authorizations derived from the authorization on `database[Inventory]` are those contained in the shadowed area; thus, because of exceptions, the user does not receive the same authorization on some objects belonging to the same subtree.

A weak authorization can also have strong exceptions. Figure 4 illustrates an example of a weak authorization on `database[Inventory]` that is overridden by a strong authorization on the class `Automobile`. Note that in Figure 3 the authorizations on the classes `Vehicle` and `Motor-Vehicle` are weak, and therefore they may be further overridden with exceptions. In the current example, however, the authorization on the class `Automobile` is strong and therefore cannot be overridden.

Combining Positive /Negative and Strong /Weak Authorizations. The combination of positive/negative authorizations and strong/weak authorizations allows a more concise and flexible representation of diverse authorization requirements. In particular, it helps to alleviate the storage requirements for authorization definitions. In fact, the combination of a weak negative authorization with strong positive authorizations allows the definition of positive authorizations as exceptions to a negative authorization at a higher level in the database granularity hierarchy. Similarly, the combination of a weak positive authorization with strong negative authorizations specifies exceptions to a positive authorization. This is illustrated in the following example.
Suppose that the class Automobile in Figure 1 has 1000 instances and that we wish to grant a read authorization to a user $s$ on all instances of the class except instance[1]. A combination of one weak positive authorization and one strong negative authorization allows this to be accomplished simply. We specify a weak read authorization on class[Automobile] for $s$ and grant to $s$ a strong negative read authorization on instance[1]. The conventional approach (unless content-based authorization is implemented [15]) would require us to specify read authorizations on 999 instances of class[Automobile]. The two situations are illustrated in Figure 5.

We may use weak authorizations to describe more complex situations. For example, $s_1$ may have a weak read authorization on the entire system[MCC], except database[Inventory] on which $s_1$ has a weak write authorization. Furthermore, $s_1$ may have a weak negative read authorization on class[Automobile] of database[Inventory] and an explicit strong write authorization on instance[1] of class[Automobile] (see Figure 6).

2.3 Formalization of the Basic Authorization Concepts

Definition 1. A positive authorization is a triplet $(s, o, a)$ with $s \in S$, $o \in O$, and $a \in A$. A negative authorization is a triplet $(s, o, \neg a)$, with $s \in S$, $o \in O$, and $a \in A$. 

ACM Transactions on Database Systems, Vol 16, No. 1, March 1991
The symbol \( \neg \) denotes negation. An authorization \( (s, o, |a|) \), with \( |a| \) denoting either \( a \) or \( \neg a \), is True (False) if \( f(s, o, a) = True \) \( f(s, o, a) = False \). Furthermore, \( f(s, o, \neg a) = True \Longleftrightarrow f(s, o, a) = False \). In the remainder of this paper, for notational simplicity, we use \( a \) in place of \( |a| \), unless it is necessary to differentiate \( a \) from \( \neg a \).

The formal definitions for the concepts of positive, negative, strong, and weak authorizations presented below represent a revision of those given in [14].

2.3.1 Strong Authorization

Definition 2. A strong authorization is a triplet \( (s, o, a) \), with \( s \in S \), \( o \in O \), \( a \in A \), and \( a \) positive or negative.

We use the term authorization to mean a strong authorization (positive or negative).

Definition 3. An authorization base \( (AB) \) is a set of strong authorizations \( (s, o, a) \), with \( s \in S \), \( o \in O \), \( a \in A \), and \( a \) positive or negative; that is,

\[
AB \subseteq S \times O \times A.
\]

Authorizations in the \( AB \) are explicit strong authorizations, and if

\[
(s, o, a) \in AB,
\]
then
\[ f(s, o, a) = \text{True}. \]

The function \( i(s, o, a) \) computes the value (True or False) of an authorization \((s, o, a)\) from the explicit authorizations in the \( AB \). \( i(s, o, a) \) returns either True or False if either the authorization \((s, o, a)\) or the authorization \((s, o, \neg a)\) can be deduced from some \((s_1, o_1, a_1)\) in the \( AB \). If neither conclusion can be deduced from the \( AB \), \( i(s, o, a) \) returns Undecided.

**Definition 4.** Function \( i(s, o, a) \) is defined as
\[
i : S \times O \times A \rightarrow \{ \text{True}, \text{False}, \text{Undecided} \}.
\]
If \((s, o, a) \in AB\), then \( i(s, o, a) = \text{True} \); else, if there exists an \((s_1, o_1, a_1) \in AB\) such that \((s_1, o_1, a_1) \rightarrow (s, o, a)\), then \( i(s, o, a) = \text{True} \); else, if there exists an \((s_1, o_1, \neg a_1) \in AB\) such that \((s_1, o_1, \neg a_1) \rightarrow (s, o, \neg a)\), then \( i(s, o, a) = \text{False} \); else \( i(s, o, a) = \text{Undecided} \).

Note that in Definition 4 we have used the implication between authorizations: \((s_1, o_1, a_1) \rightarrow (s, o, a)\). In Section 3 we formally define the rules for the implication “ \( \rightarrow \) ” for all three domains \( S \), \( O \), and \( A \).

As an example, consider the authorization in Figure 2. In that example, \((s, \text{database}[\text{Inventory}], \text{Write}) \rightarrow (s, \text{class}[\text{Automobile}], \text{Write})\), and therefore, \( i(s, \text{class}[\text{Automobile}], \text{Write}) = \text{True} \). In this case, the implication between the two authorizations occurs along the domain \( O \). Similar implication rules exist along the domains \( S \) (e.g., implication between the authorizations of an employee and the authorizations of his or her manager) and \( A \) (e.g., implication between the authorization types Read and Write).

For any authorization \((s, o, a) \in AB\), we define the scope of \((s, o, a)\), \( P(s, o, a) \) as the set of implicit authorizations \((s_1, o_1, a_1)\) such that \((s, o, a) \rightarrow (s_1, o_1, a_1)\). As a consequence, the value of function \( i \) on any authorization in the scope of \((s, o, a)\) is determined by the value of \( i \) on \((s, o, a)\). As an example, consider Figure 2 and the explicit authorization \((s, \text{database}[\text{Inventory}], \text{Write})\). The scope of this authorization along domain \( O \) is represented by all authorizations \((s, o, \text{Write})\), where \( o \) belongs to the subtree rooted at \( \text{database}[\text{Inventory}] \).

Two invariants of the \( AB \) are defined. The authorization subsystem ensures that any operation on \( AB \) always leaves \( AB \) in a state satisfying the invariants.

**Consistency of the \( AB \).** For any \((s, o, a) \in AB\), a positive or negative, if there exists an \((s_1, o_1, a_1)\) such that \((s, o, a) \rightarrow (s_1, o_1, a_1)\), then there must not exist any \((s_2, o_2, a_2) \in AB\) such that \((s_2, o_2, a_2) \rightarrow (s_1, o_1, \neg a_1)\).

**Nonredundancy of the \( AB \).** If \((s, o, a) \in AB\), a positive or negative, and \((s, o, a) \rightarrow (s_1, o_1, a_1)\), then \((s_1, o_1, a_1) \notin AB\).

The nonredundancy invariant ensures that in the \( AB \) there are no “useless” authorizations. An authorization is useless if it is implied by another (explicit) authorization. This invariant may need to be relaxed for performance.
reasons, as we will see in Section 5. The use of this invariant simplifies the definitions and the operations in the AB.

As an example, consider Figure 2 and the explicit authorization \((s, \text{database[Inventory]}, \text{Write})\). An explicit authorization \((s, \text{class[Automobile]}, \text{Write})\) is redundant since it is implied by the authorization \((s, \text{database[Inventory]}, \text{Write})\). Furthermore, an explicit authorization \((s, \text{class[Automobile]}, \neg \text{Write})\) is inconsistent with respect to \((s, \text{database[Inventory]}, \text{Write})\).

2.3.2 Weak Authorization

Definition 5. A weak (or overridable) authorization is a triplet \([s, o, a]\), with \(s \in S\), \(o \in O\), \(a \in A\), and \(a\) positive or negative.

Definition 6. A weak authorization base (WAB) is a set of weak authorizations \([s, o, a]\), with \(s \in S\), \(o \in O\), \(a \in A\), and \(a\) positive or negative:

\[\text{WAB} \subseteq S \times O \times A.\]

Weak authorizations in WAB are explicit weak authorizations.

The scope of a weak authorization \([s_1, o_1, a_1]\) \(\in\) WAB, denoted by \(\text{P}(s_1, o_1, a_1)\), is the set of weak authorizations \([s, o, a]\) that are implied by \([s_1, o_1, a_1]\) and that are not overridden by exceptions. As an example, consider Figure 3 and the explicit weak authorization \([s, \text{database[Inventory]}, \text{Write}]\). The scope of this authorization along the domain \(O\) consists of all the authorizations \([s, o, \text{Write}]\) in which \(o\) belongs to the subtree (in the database granularity hierarchy) rooted at \text{database[Inventory]} minus the authorizations \([s, \text{class[Vehicle]}, \text{Write}]\) and \([s, \text{class[MotorVehicle]}, \text{Write}]\), and all the authorizations implied by these two. In Figure 3, the scope of \([s, \text{database[Inventory]}, \text{Write}]\) is represented by all the authorizations contained in the shaded area.

As for the strong authorizations, we need to define the rules of implication between weak authorizations. The weak implication, denoted by the symbol \(\rightarrow\), is defined in terms of the scope of weak authorizations. That is, \([s_1, o_1, a_1] \rightarrow [s, o, a]\) iff \([s, o, a] \in \text{P}(s_1, o_1, a_1)\).

For example, referring to Figure 3, we have \([s, \text{database[Inventory]}, \text{Write}] \rightarrow [s, \text{instance[l]}, \text{Write}]\), since the latter belongs to the scope of the former. However, \([s, \text{database[Inventory]}, \text{Write}]\) does not imply \([s, \text{class[Vehicle]}, \text{Write}]\), since the latter does not belong to the scope of the former.

The function \(d\) determines if the value of a weak authorization \([s, o, a]\) is True or False in the WAB. \(d(s, o, a)\) is evaluated to True if the value of the weak authorization \([s, o, a]\) is True; otherwise, it is evaluated to False. Just as \(i(s, o, a)\) computes implications in the AB, \(d(s, o, a)\) computes implications in the WAB. The WAB complements the AB to enable the computation of function \(f\), which gives the value of an authorization. The function \(d(s, o, a)\) is used to determine the value of a weak authorization \([s, o, a]\), when the value of \((s, o, a)\) cannot be computed by \(i(s, o, a)\). As we will see, the value of \(d(s, o, a)\) does not include Undecided.
Definition 7. Function \( d(s, o, a) \) is defined as

\[
d: S \times O \times A \rightarrow \{True, False\}.
\]

If \([s, o, a] \in WAB\), then \( d(s, o, a) = True \); else, if there exists an \([s_1, o_1, a_1] \in WAB\) such that \([s_1, o_1, a_1] \rightarrow [s, o, a]\), then \( d(s, o, a) = True \); else, if there exists an \([s_1, o_1, ^{\neg}a_1] \in WAB\) such that \([s_1, o_1, ^{\neg}a_1] \rightarrow [s, o, ^{\neg}a]\), then \( d(s, o, a) = False \).

We define two invariants of the \( WAB \). Any operation on the \( WAB \) should bring the \( WAB \) from a state satisfying the invariants of the \( WAB \) again to a state satisfying these invariants.

**Completeness of the \( WAB \).** For any authorization \((s, o, a)\), a positive or negative, there must exist an \([s_1, o_1, a_1] \in WAB\) such that \([s_1, o_1, a_1] \rightarrow [s, o, a]\).

This invariant ensures that the entire space \( S \times O \times A \) be covered by some scope of weak authorization definition, so that function \( d \) is completely defined on \( S \times O \times A \). The completeness condition prevents function \( d \) from being evaluated to Undecided.

**Consistency of the \( WAB \).** For any \([s, o, a] \in WAB\), a positive or negative, if there exists \([s_1, o_1, a_1]\) such that \([s, o, a] \rightarrow [s_1, o_1, a_1]\), then there must not exist any \([s_2, o_2, a_2] \in WAB\) such that \([s_2, o_2, a_2] \rightarrow [s_1, o_1, ^{\neg}a_1]\).

This invariant ensures that, where the scopes of different weak authorizations overlap, these weak authorizations imply the same weak authorizations.

Note that we allow redundancy in the \( WAB \). In fact, a weak authorization, even if implied by another weak authorization (in the sense of implication "\( \rightarrow \)"), allows the modification of the scope of the second weak authorization. This is useful in solving conflicts on overlapping scopes of different weak authorizations (this problem obviously does not exist for strong authorizations).

In order to illustrate the above discussion with a significant example, we have to consider a database granularity hierarchy extended with additional semantic constructs, such as composite objects or versions. We introduce these constructs in Section 4 and include an extensive example in Appendix A. The reader will be able to understand the example better after reading Section 4; however, this should not cause any difficulties for the reader in reading the subsequent sections.

2.3.3 Coexistence of Strong and Weak Authorizations. Suppose that a write authorization is defined for a user on database[Inventory]. It is meaningless to define a negative weak write authorization for that user on class[Automobile], since it would contradict the implicit write authorization on class[Automobile]. In this case, it is clear that we need to require that an explicit weak authorization does not override explicit or implicit strong authorizations. In general, the existence of certain strong authorizations can make certain weak authorizations meaningless.
We now formally define an invariant on the coexistence of the WAB with the AB. Any operation on the AB and the WAB must always leave both the WAB and the AB in a state satisfying this invariant.

**Coexistence of the WAB and the AB.** For any \([s, o, a] \in \text{WAB}\), a positive or negative, there must not exist any \((s_1, o_1, a_1) \in \text{AB}\) such that \((s_1, o_1, a_1) \rightarrow (s, o, \neg a)\).

This invariant prefers the AB over the WAB, since we use the WAB only to complement the AB in the computation of the function \(f\) when no value can be deduced from the AB alone. The state of the WAB must always conform to the state of the AB (not vice versa); that is, a change in the WAB is allowed only if the new state of the WAB conforms to the state of the AB. If instead, a change in the AB brings it to a state that does not conform to the WAB, then the WAB must be changed to conform to the new state of the AB.

2.3.4 Operations. Basic authorization operations include checking, granting, and revoking. All the authorization operations depend on the definitions of authorization implication and weak authorization implication, which will be formalized in the remainder of this paper. Note that, since the implication rules are different for implicit strong and weak authorizations, given the different invariants on AB and WAB, the grant and revoke operations are different for strong and weak authorizations.

Note that the following rules for grant/revoke strong/weak authorizations can be regarded as the transition rules of our model, that is, rules that define the possible modifications on the AB and WAB, moving them from one consistent state to another.

**Check authorization.** To check the authorization \((s, o, a)\) means to evaluate the function \(f(s, o, a)\). The function \(f\) is defined in terms of the functions \(i\) and \(d\). First, we determine from the AB if the authorization \((s, o, a)\) is True or False. If it cannot be determined from the AB, we use the WAB to decide if the corresponding weak authorization \([s, o, a]\) is True or False.

Function \(f(s, o, a)\):
- if \(i(s, o, a) = \text{Undecided}\),
  - then \(f(s, o, a) = d(s, o, a)\);
- else \(f(s, o, a) = i(s, o, a)\).

**Grant strong authorization.** Granting the authorization \((s, o, a)\) means inserting it into the AB. This operation is implemented by the function \(g(s, o, a)\), which is evaluated to True iff the authorization is inserted in the AB, that is, if \((s, o, a)\) was not already defined in the AB, or implied. Otherwise, \(g(s, o, a)\) is evaluated to False.

**Grant weak authorization.** Granting the weak authorization \([s, o, a]\) means inserting it into the WAB. This operation is implemented by the function \(gw(s, o, a)\), which is evaluated to True iff the weak authorization is inserted in the WAB. Otherwise, it evaluates to False. The weak authoriza-
Revoke authorization. Revoking the authorization \((s, o, a)\) means deleting it from the \(AB\). This operation is implemented by the function \(r(s, o, a)\), which is evaluated to True iff the authorization was present in the \(AB\), that is, if \((s, o, a)\) was explicitly defined in the \(AB\) and not implied. Otherwise, \(r(s, o, a)\) evaluates to False.

Revoke weak authorization. Revoking the weak authorization \([s, o, a]\) means deleting it from the \(WAB\). This operation is implemented by the function \(rw(s, o, a)\), which is evaluated to True iff the authorization was present in the \(WAB\). Otherwise, \(rw(s, o, a)\) evaluates to False.

Authorization to grant or revoke authorizations. We also need rules for determining the conditions under which a subject has the authorization to grant and revoke authorizations. DB2 [4] restricts the grant privilege to only those users who hold an explicit “grant option” on the authorization. In contrast, we assume here that, if a subject \(s\) has an authorization on an object, \(s\) also has the authorization to grant the same authorization to another subject. That is, we assume for simplicity that the grant option is implicit in any authorization. This assumption can be changed by defining specific authorization types whose function is to control the grant and revoke options associated with each authorization type. For example, we may have authorization types \(W-g\) (write authorization type with grant option) or even \(W-r\) (write authorization type with revoke option) besides the normal \(W\). Other specific authorization types could be associated with an \(R\) (read authorization type). This would not change our authorization model; it will cause changes only to the authorization types in \(A\) and their implications (we discuss this possibility in Section 3.2).

The rule for checking the authorization to grant and/or revoke explicit authorizations (both for weak and strong authorizations) depends on the authorization types defined in \(A\) and, in particular, on specific authorization types introduced in \(A\) to control the granting and revocation of authorizations. Assuming no such specific authorization types, the rule states, in general terms, that subject \(s\) has the authorization to grant/revoke the explicit authorization \((s, o, a)\) if \(s\) has the same positive authorization. That is, \(s\) has the authorization to grant/revoke \((s, o, a)\) if \(f(s, o, a) = True\). The same rule is used for checking the authorization to grant/revoke a weak authorization: Subject \(s\) has the authorization to grant/revoke \([s, o, a]\) iff \(s\) has the authorization to grant/revoke \((s, o, a)\).

3. IMPLICATION RULES

As we discussed earlier, implicit authorizations can occur along any of the authorization dimensions. In this section we define the rules for the implication (denoted by the symbol “\(\rightarrow\)”) between authorizations along each of the three dimensions. These rules are used for deducing implicit authorizations from explicitly defined strong authorizations. Section 3.1, on authorization
subjects, is a revision of the discussion in [14]. Sections 3.2 and 3.3, on authorization types and authorization objects, represent major extensions and changes to the model of [14].

3.1 Authorization Subjects

To reduce the number of authorization subjects that need to be involved in explicit definitions of authorizations, we group users according to their roles and associate authorizations with the roles. A user may belong to more than one role. The roles form a rooted directed acyclic graph called a role lattice (RL).

A node of the graph represents a role, and a directed arc from role A to role B indicates that the authorizations for role A subsume the authorizations for role B. A role can have multiple parent roles; all users who belong to the parents of a role also have all authorizations of the role. The root of the RL is the superuser role, which is the ancestor role of all other roles in the RL. An example role lattice is shown in Figure 7.

Note that the topmost role (i.e., the superuser), as well as the bottommost role, is required only for completeness of the role lattice (and to initialize the RL). The superuser is a powerful, and potentially dangerous, role. If it is not wanted, no system user may be assigned to this role.

The rule for checking authorization of users is as follows: A user has authorization a on o if there exists a role s such that \( s, o, a \) = True and the user belongs to s.

Definition 8. The notation \( s_i > s_j \), where \( s_i \in S \) and \( s_j \in S \), means that there is an implication link from \( s_i \) to \( s_j \) in the RL. The notation \( s_i \geq s_j \), where \( s_i \in S \) and \( s_j \in S \), means that either \( s_i = s_j \) or \( s_i > s_j \), or there exist \( s_1, \ldots, s_n \) in S, such that \( s_i > s_1 > \cdots > s_n > s_j \).

We have the following rule and corollary for the implicit authorizations on roles directly or indirectly connected in the RL:

Rule 1. For any \( o \in O \) and \( a_n \in A \), if \( s_i \geq s_j \) then \( (s_j, o, a_n) \rightarrow (s_i, o, a_n) \).
Corollary 1 (This corollary is derived from rule 1 and applies to negative authorizations.) For any $o \in O$ and $a_n \in A$, if $s, \geq s, \text{then} (s,, o, \neg a_n) \rightarrow (s, o, \neg a_n)$.

Corollary 1 is derived from Rule 1 by observing that in the predicate calculus $(p \rightarrow p') \leftrightarrow (\neg p' \rightarrow \neg p)$ for $p$ and $p'$ predicates.

Rule 1 states that, if a role has an authorization on an object, all roles that are higher in the RL implicitly receive the same authorization. Conversely, if a role is denied an authorization on an object, all roles that are lower in the RL are implicitly denied the authorization.

For example, $(S/W\text{-project, class[Automobile], W}) \rightarrow (Application\text{-project, class[Automobile], W})$ because Application-project $> S/W\text{-project}$. Conversely, $(Application\text{-project, class[Automobile], }\neg W) \rightarrow (S/W\text{-project, class[Automobile], }\neg W)$.

3.2 Authorization Types

The authorization types are organized in an Authorization Type Lattice (ATL). The ATL is a rooted directed acyclic graph in which each node is an authorization type. An arc from node $a_i$ to node $a_j$ indicates that authorization type $a_i$ implies authorization type $a_j$.

Definition 9. The notation $a_i > a_j$, where $a_i \in A$ and $a_j \in A$, means that there is an implication link from $a_i$ to $a_j$ in the ATL. The notation $a_i \geq a_j$, where $a_i \in A$ and $a_j \in A$, means that either $a_i = a_j$ or $a_i > a_j$, or there exist $a_1, \ldots, a_n$ in $A$, such that $a_i > a_1 > \cdot \cdot \cdot > a_n > a_j$.

For simplicity of presentation, we assume for now that any authorization type may be associated with any object type. We relax this assumption in the next section.

Rule 2. For any $s \in S$ and $o \in O$, if $a_i \geq a$, then $(s, o, a_i) \rightarrow (s, o, a)$.

Corollary 2. For any $s \in S$ and $o \in O$, if $a_i \geq a_j$ then $(s, o, \neg a_i) \rightarrow (s, o, \neg a_j)$.

The authorization types that we consider in this paper include R (Read), W (Write), G (Generate), and RD (Read Definition). Therefore $A = (W, R, G, RD)$. A generate authorization is used to create an authorization object, and a read-definition authorization is used to read the definition of an authorization object. The following implication holds among authorization types in $A$: $W > R$, $W > G$, $G > RD$, $R > RD$. The corresponding ATL is illustrated in Figure 8. By applying the transitivity property, we have
W ≥ RD. Note that a generate authorization on a class implies a read definition authorization on the same class, since a user must be able to read the class definition in order to create instances of that class. However, a generate authorization on a class does not imply a write (or read) authorization on instances of that class. This is reasonable since the fact that a subject s can create instances of a class does not imply that s can modify or read all instances of that class. s can only modify or read the instances that he has created or for which s has been granted an authorization.

For example, (S/W-project, class[Automobile], W) → (S/W-project, class [Automobile], R) because W > R. Conversely, (S/W-project, class [Automobile], ¬R) → (S/W-project, class[Automobile], ¬W).

Note that the current ATL is not the only ATL possible in the model. Suppose, for example, that we wish to introduce specific authorization types to distinguish the grant option associated with any authorization type. This means introducing W-g, R-g, G-g, and RD-g in A. It is also necessary to specify implications of these new authorization types: W-g > R-g, W-g > G-g, G-g > RD-g, R-g > RD-g, W-g > W, R-g > R, G-g > G, and RD-g > RD. The corresponding ATL is illustrated in Figure 9. Other authorization types are introduced in the remainder of this paper.

3.3 Authorization Objects

 Implicit authorizations for the domain O of authorization objects have a greater potential for savings in storage space than those for authorization types and authorization subjects, because the number of objects in a database is in general much larger than the number of subjects or authorization types.

In Section 3.3.1 we first outline the concepts of the Authorization Object Schema (AOS) and Authorization Object Lattice (AOL). The AOS is the schema for the authorization object types, while the AOL is an instance of the AOS for a given database. In Sections 3.3.2–3.3.4 we develop a formal model of implicit authorization using the AOS and AOL. The model we describe here represents significant changes to that developed in [14]. The new model takes into account the fact that not all authorization types are defined for all objects; furthermore, it includes a formal definition of implicit authorizations along the AOL.

3.3.1 Authorization Object Schema (AOS) and Authorization Object Lattice (AOL). In a database granularity hierarchy, such as the one in Figure 1, each node represents at least two types of information that may be
separately authorized: about the node itself and about the set of nodes at the next-lower level. In other words, each node in a database granularity hierarchy is associated with more than one authorization object. For example, an operation on a class in Figure 1 may be applied on the class object or on the set of instances of the class. The class node needs to be split into two different authorization objects: a class authorization object and a setof-instances authorization object. Figure 10 shows the authorization objects for a database.

The reason we define authorization objects, rather than using the database objects described by a database granularity hierarchy, is to minimize the number of authorization types. Without the concept of authorization objects, we would have to double the set of authorization types: one for a node of the database granularity hierarchy, and another for the next-lower-level nodes. For example, if we did not split the class node into two authorization objects, we would need six authorization types for the class node: write class object, read class object, read class definition, write all instances of the class, read all instances of the class, and generate instances of the class. By splitting the class node into the authorization objects class and setof-instances, we can keep the authorization types to read, read definition, write, and generate. For example, a write authorization on class[Automobile] authorization object means that the class definition can be updated. A write authorization on the setof-instances[Automobile] authorization object means that any of the instances can be updated. This principle is not valid for the user interface: At this level authorizations can be presented in a more intuitive way.

We call the structure in Figure 10 an Authorization Object Lattice (AOL). An AOL is a rooted directed acyclic graph in which each node is an authorization object. A directed arc in an AOL will be called an implication link between authorization objects. An implication link defines an implicit authorization between two nodes: an arc from node A to node B indicates that an authorization on node A implies an authorization on node B. The nodes

Fig. 10. Example authorization object lattice.
shown in *italics* in Figure 10 are authorization objects that may have implication links to a set of authorization objects at the next lower level. For example, the `setofinstances[Automobile]` authorization object may have implication links to all instance authorization objects of class `Automobile`. The nodes shown in nonitalics in Figure 10 are restricted to one implication link to the node at the next lower level. For example, the class `Automobile` authorization object has an implication link to only one `setofinstances` authorization object. (In Section 4 we introduce additional types of implication links.)

Each node in an AOL belongs to one, and only one, *authorization object type*. The authorization object types collectively constitute the schema for the authorization objects. We call this schema the *Authorization Object Schema* (AOS). Figure 11 shows a diagram that represents the schema for the authorization objects of Figure 10. Each node in an AOS is an authorization object type, and a directed arc represents an implication link between a pair of instances of two authorization types. The AOL is a virtual structure that can be derived from the AOS and the state of the database. For example, the implication link between the authorization object `setofinstances[Automobile]` and the authorization object instance[1] is derived from the link between the `SetofInstances` node and the `Instance` node of the AOS of Figure 11 and the existence of the class `Automobile` and an `Automobile` instance in the actual database. We only use the AOL when computing implicit authorizations.

3.3.2 Association of Authorization Types with Authorization Objects. We must now recognize that each node in the AOS is associated with the authorization types that are semantically meaningful for objects of the type represented by the node. Some authorization types, such as R and W, are defined for all nodes; whereas others, such as G, are only meaningful for some of the nodes.

We specify the association of authorization types with authorization object types in terms of an *Authorization Association Matrix* (AAM). Figure 12 shows an example AAM for the AOS shown in Figure 11. The columns
represent authorization types, and the rows, authorization object types. The
\([AT, a]_{i,j}\)th entry is True (T) if the authorization type \(a\) is defined for objects
of type \(AT\), and False (F) otherwise.

We require that all authorizations stored in the \(AB\) (and \(WAB\)) be seman-
tically meaningful. This is formalized by the following notion of correctness.
All rules and corollaries presented earlier must be modified by imposing the
following notion of correctness: For any \((s, o, a) \in AB\) \((WAB)\), \(c(o, a) = True\).

The function \(c\) is defined as follows: \(c: O \times A \rightarrow (True, False)\). Given a pair
\((o, a)\), the function \(c\) tells us whether an authorization type \(a\) is defined for
an object \(o\). This function is computed by determining the authorization
object type of \(o\) and then by accessing the \(AAM\).

We now consider the interplay of the authorization types in \(A\) with the
authorization objects in \(O\). We can observe the different ways in which
authorization types propagate through authorization objects, along the impli-
cation links of the \(AOL\). Some authorization types, such as \(W\) and \(R\),
propagate top down along the \(AOL\); that is, when defined on an object \(o\), they
imply the existence of the same authorization type on the set of authorization
objects at the next lower level on the database granularity hierarchy. For
example, consider a read authorization on a class. A read authorization is an
authorization to read all information about the class, including any aggre-
gate property of the instances of the class; furthermore, it implies a read
authorization on all instances of the class. We group all such authorization
types into a subset of \(A\) named \(A_{down}\); \(A_{down}\) includes \(R\) and \(W\).

Some authorization types propagate through authorization objects bottom
up along the implication links of the \(AOL\). For example, consider the
authorization to read an instance. Reading an instance implies reading its
definition (i.e., \(R > RD\)). However, since the instance definition is not au-
tonomously but is given by the definition of the class to which the instance
belongs, \(RD\) on the instance implies \(RD\) on its class. Similarly, \(RD\) on a class
implies \(RD\) on the database to which the class belongs, and not vice versa.
We group all such authorization types into a subset of \(A\) named \(A_{up}\); \(A_{up}\)
includes \(RD\).
Some authorization types do not propagate through authorization objects. For example, a subject $s$ with a generate authorization on a database can create a new class in the database. However, this authorization does not imply a generate authorization on instances of other classes in the database; $s$ can only create instances of the classes $s$ has created or for which $s$ has been granted an authorization. We group all such authorization types into a subset of $A$ named $A.nil$; $A.nil$ includes $G$.

### 3.3.3 Implication Links between Authorization Objects

In this section we describe the semantics of the implication links between authorization objects.

**Definition 10.** The notation $o_i > o_j$, where $o_i \in O$ and $o_j \in O$, means that there is an implication link from $o_i$ to $o_j$ in the AOL. The notation $o_i \geq o_j$, where $o_i \in O$ and $o_j \in O$, means that either $o_i = o_j$ or $o_i > o_j$, or there exist $o_1, \ldots, o_n$ in $O$, such that $o_i > o_1 > \cdots > o_n > o_j$.

The authorizations implied in the implication links between authorization objects are defined in the following rule and corollary:

**Rule 3.** For any $s \in S$ and any $a_n \in A.down$, if $o_i \geq o_j$, $c(o_j, a_n) = T$, and $c(o_i, a_n) = T$ then $(s, o_i, a_n) \rightarrow (s, o_j, a_n)$. For any $s \in S$ and any $a_n \in A.up$, if $o_i \geq o_j$, $c(o_j, a_n) = T$, and $c(o_i, a_n) = T$ then $(s, o_j, a_n) \rightarrow (s, o_i, a_n)$.

**COROLLARY 3 (this corollary is derived from Rule 3 and applies to negative authorizations).** For any $s \in S$ and any $a_n \in A.down$, if $o_i \geq o_j$, $c(o_j, a_n) = T$, and $c(o_i, a_n) = T$ then $(s, o_i, \neg a_n) \rightarrow (s, o_j, \neg a_n)$. For any $s \in S$ and any $a_n \in A.up$, if $o_i \geq o_j$, $c(o_j, a_n) = T$, and $c(o_i, a_n) = T$ then $(s, o_j, \neg a_n) \rightarrow (s, o_i, \neg a_n)$.

In Rule 3 and Corollary 3 the preconditions $c(o_j, a_n) = T$ and $c(o_i, a_n) = T$ state that only authorizations defined on $o_j$ and $o_i$ can be derived from other objects through the implication links.

We now revisit the differences among $A.down$, $A.up$, and $A.nil$ in terms of the implicit authorizations on authorization objects. Since negative authorizations propagate up the AOL (whereas positive authorizations propagate down the AOL), it is useful to have a specific authorization type in $A.up$ that defines the negation of an authorization type in $A.down$. For example, if we wish to define the negation of a read authorization on Class[Automobile] and all its instances, we need to define an authorization $\neg RD$ on Class[Automobile]. This implies $\neg R$ on all instances of Class[Automobile], whereas a $\neg R$ authorization on Class[Automobile] would not imply a $\neg R$ on its instances. Similarly, with the write authorization (W), we need to introduce a new authorization type WA (Write Any) in $A.up$. Again, whereas a $\neg W$ authorization on Class[Automobile] does not imply a $\neg W$ on instances of Class[Automobile], $\neg WA$ does. Therefore, the set $A$ is considered as consisting of $A.down = \{W, R\}$, $A.up = \{WA, RD\}$ and $A.nil = \{G\}$. The ATL defining the implication between these authorization types is illustrated in Figure 13.

Figures 14a and b illustrate the authorizations implied by an explicit authorization. In the example, we use a graphical representation (devised in...
Fig. 13. Extended authorization type lattice.

[14]) for the implicit authorizations between authorization objects connected by implication links. In the figure the nodes at the same horizontal level indicate different authorization types for an authorization object. The directed arcs indicate implied authorizations. We consider the authorization types shown in Figure 13 and the implication links between the authorization objects database[Inventory], setof.instances[Automobile], and class[Automobile]. Figure 14a shows that a positive W authorization on the class Automobile implies an R (and RD) authorization on the class and W authorizations on all Automobile instances. Figure 14b shows that a negative RD authorization on the class Automobile implies a negative R and a negative W authorization on the class Automobile and on all the instances of this class.

3.3.4 Rules for Computing Implicit Strong Authorizations. From the definitions of implicit authorizations for authorization subjects in S, authorization types in A, and authorization objects in O, the complete set of rules and corollaries is derived for computing implicit authorizations from the explicit authorizations stored in the AB. The rules and corollaries are presented in Appendix B. Similar rules for computing implicit weak authorizations are derived and presented in Appendix C.

Below we illustrate the use of the combined implication rules along the three authorization dimensions. Using the Role Lattice of Figure 7 and the Authorization Object Lattice of Figure 10, we show the interplay of implicit authorizations between authorization subjects and objects.

Suppose that the following authorization is defined in the AB: (S/W-project, database[Inventory], W); and that the authorization (OS-project, instance[1] of class[Automobile], R) needs to be checked. Along domain S, we have that OS-project > S/W-project: along domain O, we have that database[Inventory] > Class[Automobile] > instance[1] of Class[Automobile]; and along domain A, we have that W > R, with W e A.down. Therefore, by applying Rules 1, 2, and 3, we get (S/W-project, database[Inventory], W) -> (OS-project, instance[1] of class[Automobile], R), and therefore, f(OS-project, instance[1] of class[Automobile], R) = True.

To illustrate the implications between negative authorizations, suppose that the following negative authorization is defined in the AB: (O/S-project, Class[Automobile], −RD); and that the authorization (employee, instance[1] of class[Automobile], W) needs to be checked. We have that along domain S, OS-project > S/W-project > employee; along domain O, Class

A Model of Authorization for Next-Generation Database Systems

Fig. 14. (a) Implications of a positive authorization \((S, \text{class[Auto]}, W)\); (b) implications of a negative authorization \((e, \text{class[Auto]}, \neg RD)\).

[Automobile] > instance[1] of Class[Automobile]; and along domain A, \(W > R > RD\), with \(RD \in A_{up}\). By applying Corollaries 1, 2, and 3, we have \((O/S-	ext{project}, \text{Class[Automobile]}, \neg RD) \rightarrow (\text{employee, instance[1] of class [Automobile]}, \neg W)\), and therefore, \(f(\text{employee, instance[1] of class [Automobile]}, \neg W) = True\) and \(f(\text{employee, instance[1] of class [Automobile], W}) = False\).

Below we illustrate the grant of an authorization. On the basis of the authorization \((S/W-	ext{project, database[Inventory], W})\) and the authorizations it implies, the following sequence of authorization operations is valid:

accounts-manager issues: \(g(\text{accounts-employee, class[Automobile], W})\)
accounts-clerk issues: \(g(\text{clerk, setof-instances[Automobile], R})\)
accounts-clerk issues: \(g(\text{clerk, instance[1] of class Automobile, } \neg W)\).

Note that the first grant operation gives rise to the implicit authorization
(accounts-clerk, setof-instances[Automobile], R), which in turn makes the next
two grant operations possible.

Let us now consider the following sequence of grant operations:

accounts-manager issues: \( g(S/W-project, \text{class}[Automobile], W) \)
S/W-project issues: \( g(H/W-project, \text{setof-instances}[Automobile], W) \)
accounts-manager: \( g(H/W-project, \text{instance}[1] \text{of class Automobile}, R) \).

The third grant operation is rejected, since it would be inconsistent with the
second authorization \( (H/W-project, \text{setof-instances}[Automobile], W) \). A possible
line of action is to transform the strong authorization \( (H/W-project, \text{setof-}
instances[Automobile], W) \) to a weak authorization, which could admit excep-
tions such as the strong negative authorization \( (H/W-project, \text{instance}[1] \text{of}
\text{class Automobile}, \neg R) \). This can be accomplished with the following sequence
of operations:

accounts-recv-mgr: \( r(H/W-project, \text{setof-instances}[Automobile], W) \)
accounts-recv-mgr: \( gw(H/W-project, \text{setof-instances}[Automobile], W) \)
accounts-recv-mgr: \( g(H/W-project, \text{instance}[1] \text{of Automobile}, R) \).

4. IMPLICIT AUTHORIZATIONS FOR OBJECT-ORIENTED AND SEMANTIC
MODELING CONCEPTS

We now extend the model of authorization developed thus far to key data
modeling concepts found in object-oriented databases and semantic databases,
including composite objects and versions. The bases of the extension of our
model of authorization, to account for the semantics of objects, are actually
object-oriented concepts found in most object-oriented systems. To make the
discussions of implicit authorizations on composite objects and versions con-
crete, we make explicit use of the semantics of composite objects and versions
as they are supported in ORION [1, 12]. However, insofar as a composite
object is in essence a set of objects related through the \( \text{IS-PART-OF} \) relations-
ships, and a versioned object is similarly a set of objects related through the
\( \text{IS-VERSION-OF} \) relationships, our results are applicable to other models of
composite objects and versions with different semantics.

4.1 Properties of a Class

In order to apply the model of authorization to the object-oriented data
modeling concepts, we introduce additional authorization object types in the
AOS. Figure 15 is a complete AOS for the authorization object types
supported in ORION. The bold arcs represent the implication links between
authorization objects described in Section 3. The nonbold arcs represent the
implication links for the new authorization objects that have been added for
the new semantic modeling concepts. These new authorization objects are
connected through implication links. We have chosen to keep the number of
authorization types small and to use them also for the new authorization
objects. One unpleasant consequence of this decision is that the authorization
types may be overloaded; that is, the same authorization type may be used
for different authorization objects with somewhat different semantics. While
this simplifies the ATL, and therefore the design of the authorization mechanism, it may be less than intuitive to the user. However, we can provide a user interface that allows a more intuitive expression of the semantics of the authorization types.

Database systems usually define authorizations for the schema entities, such as classes, attributes, and indexes. For example, DB2 [4] supports various authorizations on relations, attributes, and indexes. An object-oriented database schema introduces an additional schema entity, namely, methods. Also, since classes are themselves objects, an object-oriented database schema distinguished between the instance attributes that characterize all instances of the class and the class attributes that characterize the class itself as an object.

To describe any specific authorization on the attributes of a class, we introduce the authorization objects setof-Attributes and Attribute (see Figure 15). A W (or R) authorization for a subject on a particular attribute means that the subject can update (or read) the values of this attribute in all instances of the class. The implicit authorization on the corresponding attribute values of all instances is reflected in the implication link from the node Attribute to the node Attribute-Value in the AOS. Similarly, we model authorizations on methods by introducing the authorization objects setof-Methods and Method. Since most object-oriented systems do not allow meth-
ods to be specified for instances, there is no implication link from the node Method to the Instance node.

The authorizations on attributes and methods are restricted to the extensional aspects of a class, that is, values of attributes in instances or invocation of methods on instances. They do not affect the intensional aspects of a class, that is, the definition of attributes and methods; these are controlled by a W authorization on the node Class (i.e., an authorization to change the class definition). We assume that an R authorization on Methods acts as an authorization to execute the corresponding method. All the operations that are performed during the execution of the method must be checked further as necessary. For example, if, during the execution of a method invoked by a subject s, an attempt is made to update a particular attribute value of an instance, the authorization for s to update the attribute value needs to be verified.

Authorizations on indexes are modeled by the authorization type W and by the authorization objects Setof-indexes (e.g., to drop all indexes) and Index (e.g., to drop a particular index). A G authorization on the authorization object type setof-indexes is used for the authorization to create an index on a class. Although possible, it does not seem useful to control the use of indexes in query processing by individual users using the R authorization on Index. In fact, an index is a resource that is created to speed up database access and as such can be accessed only by the system. Therefore, there are no operations available at the user interface level other than create and drop an index. Operations such as read or modify an index cannot be invoked by the user, and therefore, there is no need for the corresponding authorization types.

Figure 16 shows the AAM for authorization types associated with authorization objects for the properties of a class. The implication rules, defined previously, can be applied to objects in the AOL. As an example, let us consider the G authorization type defined for the setof-Instances authorization object type. The implication rules tell us that the G authorization implies an RD authorization on the setof-Instances. This authorization in
turn implies an RD authorization on the Class authorization object type, that is, the authorization to read the class definition. Therefore, the result of applying the implication rules is that a user with a G authorization on the setofInstances implicitly receives the authorization to read the class definition, which is meaningful from the semantic point of view since a user would not be able to create instances of a class without knowing their definition.

4.2 Class Hierarchy

Figure 17 shows an example class hierarchy. The class Automobile is a subclass of both the class 4-Wheel-Vehicle and the class Motor-Vehicle, both of which are subclasses of the class Vehicle. The class Automobile inherits attributes and methods from both of its superclasses. An object-oriented model encourages the specialization of existing classes to form new classes and thereby promotes the reuse of existing specifications (and designs). As such, care must be taken not to have the authorization mechanism discourage this.

There are two positions that can be taken concerning authorization on the instances of a subclass. The first position is that the creator of a class should have no implicit authorizations on the instances of a subclass derived from the class. Therefore, the creator of the class Vehicle should not be able to update or read instances of the class Automobile unless explicitly given that authorization by the creator of the class Automobile (or other authorized users). This position encourages the reuse of existing classes without diminishing privacy. However, a query whose scope of access is a class and its subclasses will only be evaluated against those classes for which the user issuing the query has a read authorization.

The second position is that the creator of a class should have implicit authorizations on instances of a subclass. This means that the creator of the class Vehicle should be able to update and read instances of the class Automobile. This means that a query whose scope of access is a class and a class subhierarchy rooted at the class will be evaluated against the class and all its subclasses.

We have elected to adopt the first position as the default and to support the second as a user option. Figure 15 includes a new authorization object Class-Hierarchy, which we introduce to support authorizations on a class hierarchy. The authorization types that are defined on this new object type are R and W. An R (or a W) authorization on a Class-Hierarchy object implies
read (or write) authorizations on all of the classes in the class hierarchy. Our choice is motivated by the observation that in the second position a user wishing to derive a subclass from another class would not have any privacy on the instances of the subclass. Therefore, when privacy is important, the user would not be able to reuse the existing classes to define his or her own classes; that is, the second position would prevent the users from taking advantage of inheritance.

When multiple inheritance is supported in the data model, implicit authorizations along the class hierarchy may give rise to conflicts. These are handled by using the mechanisms we have introduced in the previous sections. As an example, if a user receives an R authorization on the Class-Hierarchy object rooted at the class Motor-Vehicle, the user implicitly receives an R authorization on the classes Motor-Vehicle and Automobile. Let us now assume that the user receives a negative R authorization on the Class-Hierarchy object rooted at the class 4-Wheel-Vehicle. This authorization implies a negative R authorization on the class Automobile, and therefore, it conflicts with the authorization on the Class-Hierarchy object rooted at the class Motor-Vehicle. Therefore, this second authorization is rejected, unless the first authorization is weak and therefore can be overridden.

We would like to use the authorization mechanism also to control the creation of subclasses of a class. For this reason, we introduce in the set $A$ the authorization type $SG$ for the authorization object Class. An $SG$ (subclass generate) authorization on a class is an authorization to create subclasses of the class. The $SG$ authorization belongs to $A.nil$, and the ordering between the $SG$ and other authorization types previously introduced is the following:

\[ W \geq SG \quad \text{and} \quad SG \geq RD. \]

The first ordering states that a user with a $W$ authorization on a class receives an implicit authorization to derive subclasses. The second states that a user with an authorization to create a subclass from a given class $C$ implicitly receives an $RD$ authorization; that is, the user can read the class definition of $C$, which is needed for the user to derive a new class from $C$.

Suppose that the Inventory database contains the classes Vehicle, 4-Wheel-Vehicle, and Motor-Vehicle, as shown in Figure 17. A subject $S$ wishes to create a new class Automobile, which would inherit properties from both 4-Wheel-Vehicle and Motor-Vehicle. Of course, $S$ needs a $G$ authorization on database[Inventory] to create the class Automobile. $S$ also needs an $SG$ authorization on class[4-Wheel-Vehicle] and class[Motor-Vehicle]. $S$ will then have a $W$ authorization on class[Automobile] and, hence, an implicit $SG$ authorization on class[Automobile]. Therefore, $S$ may now create new subclasses of Automobile.

Figure 18 illustrates an AOL for an example database where the class Pickup has been created as a subclass of the class Automobile. When a subject, say, $S_1$, requested to create the class Automobile, $S_1$ was required to have an explicit or implicit $G$ authorization on database[Inventory]. $S_1$ will then have a $W$ authorization on class[Automobile] and class-hierarchy[Automobile]. This gives $S_1$ the authorization to read and update all instances of
the class Automobile and the instances of any subclasses of Automobile that may later be created by S1.

4.3 Composite Objects

A composite object is a part hierarchy that captures the IS-PART-OF relationship between an object and its parent [11]. The instances belong to a number of different component classes. A composite hierarchy is the part of a class-composition hierarchy on which the IS-PART-OF semantics is imposed. The root of a composite hierarchy is a composite class, and all other classes on the hierarchy are component classes. If an object and its component objects are created by the same user, there should be no conflict. In general, however, different users may create these instances.

A systematic mechanism is presented in [14] for defining implicit authorizations for a collection of objects that comprise a composite object. The mechanism also detects authorization conflicts that may arise when authorizations are defined on individual objects that make up a composite object. The mechanism uses the implication links along with a few new authorization objects to support implicit authorizations on composite objects. The new authorization objects are setof-Comp-Instances, Comp-Instance, setof-Comp-Attr-Values. Figure 19 illustrates the AOL for an example database that contains a composite object.

We now show how a composite object may be used as a unit of authorization by applying the concept of implicit authorization. An authorization on a composite class C implies the same authorization on all instances of C and on
all objects that are components of the instances of C. For example, consider a composite hierarchy consisting of the classes Automobile, Autobody, and AutoDrivetrain. If a user is granted an R authorization on the class Automobile, the user implicitly receives the same authorization on all instances of Automobile, and all instances of Autobody and AutoDrivetrain that are components of the instances of Automobile. Note, however, that the authorization on Automobile does not imply the same authorization on all instances of Autobody and AutoDrivetrain, since not all instances of Autobody and AutoDrivetrain may be components of Automobile. Furthermore, because of negative authorizations, a new authorization issued on a component class may conflict with a previously granted authorization on the class. In this case, the authorization subsystem must reject the new authorization.

Similarly, an authorization on a composite object implies the same authorization on each component of the composite object. For example, if a user is
granted an R authorization on the root of the composite object comp-instance[1] in Figure 19, the user implicitly receives an R authorization on each of the component objects, instance[1], comp-instance[2], and comp-instance[3]. Implicit authorizations on comp-instance[2] and comp-instance[3] in turn imply R authorizations on instance[1] and instance[3]. Therefore, the user receives implicit authorizations on all instances that are components of comp-instance[1]. Again, if a new authorization issued conflicts with an existing authorization, the new authorization is rejected.

If an instance is a component of more than one composite object, a user can receive more than one implicit authorization on that instance. For example, consider the composite objects in Figure 23a (see also Appendix A). If a user receives a read authorization on the composite object rooted at comp-instance[1], the user implicitly receives an R authorization on comp-instance[4]. If the user is later granted an R authorization on the composite object rooted at comp-instance[3], the user again receives an implicit authorization on comp-instance[4].

Negative authorizations give rise to conflicts among implicit authorizations on objects that are components of more than one composite object. For example, if the user is granted a negative strong R authorization on the composite object rooted at comp-instance[4], it will conflict with the (positive) implicit authorization the user received from comp-instance[1]. In this case, the negative authorization can be granted only if the R authorization on the composite object rooted at comp-instance[1] is a weak authorization (and, therefore, it can be overridden).

4.4 Versions

There are two ways to bind an object with a versioned object: static and dynamic [2]. In static binding, the first object directly references the second object. In dynamic binding, the first object does not directly reference the second object, but instead references a generic instance. A generic instance maintains the history of derivation of versions of an object. When a generic instance receives a message, the message is forwarded to one of these versions, which has been designated as the default version.

It is useful to distinguish two types of versions, transient and working, on the basis of the types of operation that may be allowed on them. A transient version, for example, instance[4] in Figure 20, may be modified or deleted by the user who created it; whereas a working version may be deleted but not updated. However, a new version cannot be derived from a transient version. A transient version must first be promoted to a working version before a new version may be derived from it.

We would like to specify authorizations on a versioned object and on individual versions of the object. As we have done for composite objects, we can accomplish this using implication links and a few new authorization objects, as shown in Figure 15. For all of these new authorization objects, the R and W authorization types are defined. An authorization on a setofgeneric-instances node implies the same authorization on all generic instances of the class. A W authorization on a generic instance allows the user
to modify the generic instance itself (e.g., change the default version); it also
implies authorization on the version objects described by the generic in-
stance. An R authorization on a generic instance means that the user has the
same authorization on all versions. A W authorization on the setof-versions
node implies the same authorization on the two versions described by the
generic instance. The W authorization on the node setof-versions is also the
authorization to create a new version, of the specific generic instance, from a
working version of the instance.

Figure 20 illustrates an AOL for a database that contains versioned
objects. In this example, two generic instances have been created in the
database, and each generic instance is related to two versions. The nodes
generic-instance[1] and generic-instance[2] in the AOL represent the two
genetic instances in the database. A W authorization on generic-instance[1]
allows the user to modify the generic instance itself and implies authoriza-
tion on the two versions represented by generic-instance[1]. An authorization
on the setof-generic-instances[Automobile] node implies the same authoriza-
tion on all of the generic instances of the class Automobile. An authorization
on the setof-versions[1] node, represented by the generic-instance[1] node,
implies the same authorization on the two versions represented by generic-instance[1]. In this way, we are able to specify instance-level authorizations over a potentially large number of selected instances using a single explicit authorization. A W authorization is required on the transient version instance[4] in order to promote it to a working version. A new version can then be derived from this working version. If subject $S$ wishes to derive a new version from instance[4], $S$ must have a W authorization on setof:versions[1].

5. IMPLEMENTATION CONSIDERATIONS

In this section we discuss implementation strategies for the authorization model described in the previous sections. We describe the data structures needed to implement the role lattice, and the authorization objects, namely, the $AB$ and the $WAB$. We also discuss access strategies to these data structures. These data structures are protected by allowing only the system to access them.

5.1 Role Lattice

Two data structures are needed to implement the role lattice. One maintains the role lattice. For each node in the role lattice, the list of its children and parents is recorded. This data structure is defined as a class $Roles$, with attributes $Role$, $Children$, and $Parents$, as well as methods $RetrieveDescendants$ and $RetrieveAncestors$. Given a role $S.i$, the method $RetrieveDescendants$ determines all roles $S.j$ that are direct and indirect descendants of role $S.i$; the method is used when checking positive authorizations. The method $RetrieveAncestors$ determines, for a role $S.i$, all roles $S.j$ that are direct and indirect ancestors of role $S.i$; the method is invoked when checking negative authorizations.

A second data structure associates users with their roles. A user $U$ may belong to more than one role. This data structure is specified as a class $UserRole$, with attributes $UserName$ and $Role$.

5.2 $AB$ and $WAB$

Authorizations defined in the $AB$ and $WAB$ are stored in several system catalogs. Each of the catalogs maintains authorizations on objects of a particular authorization object type. Maintaining several catalogs allows efficient authorization checking for authorization objects belonging to higher levels (close to the root) in the AOS, even when the granularity of authorization is very small. For example, allowing instance-level authorizations does not impact the time needed to check authorizations at the database level, since authorizations on databases and those on instances are stored in separate catalogs. Furthermore, this approach is cleaner and allows the authorization mechanism to be more easily modified and extended. Figure 21 shows the catalogs used for the authorization object types in our authorization model. As an example, we represent catalog $ClassAB$ as a class $ClassAB$, with attributes $Subject$, $ClassID$, and $AuthorizationType$. 

The authorization catalogs are used to store both strong and weak authorizations. One issue concerning weak authorizations is the computation of their scope. In particular, exceptions to a given weak authorization may occur on objects several levels down in the AOS from the object on which the weak authorization holds. For example, a subject may have a positive weak R authorization on a database and a negative R authorization on a composite instance. We associate an exception flag with each weak authorization. If the flag is on, additional accesses to other catalogs may be needed. Figure 22 summarizes the catalog access strategies for each authorization category. In the figure the arrow from one catalog to another indicates that the latter catalog must be accessed if the authorization is not found or a weak authorization with exceptions is found in the first catalog.

Reducant Storage of Authorizations. As described earlier, checking an authorization may require determining whether the authorization may be derived from the authorizations stored in the authorization catalogs. One important question is whether implicit authorizations must be evaluated each time an authorization is checked, or if they should be stored as redundant authorizations.

Checking an authorization for a subject S on object O, j requires two steps:

1. determine an object O, i such that O, i \subseteq O, j, and
2. check the authorization of subject S on O, i.

The cost of the first step depends on the authorization object types to which O, i and O, j belong. For example, in the case of classes and instances, this is
easy: Given an instance we can determine its class easily from the object identifier of the instances. For composite objects, this is more difficult. When a user receives an authorization on a composite object, he or she implicitly receives the same authorization on the component objects. Therefore, when the user accesses a component object through its root object, there is no need to check authorization for each component object. However, if the user accesses any component object directly (i.e., without accessing the root object), the system should first find the composite object to which the component object belongs and then check the authorization on the composite object. If the component objects do not include the identifier of their root object, step (1) can be very expensive.

Therefore, different redundancy strategies are suitable for different authorization object types. They are denoted as follows: Given authorization object types $OT$ and $OT'$ such that $OT \rightarrow OT'$ in the AOS,

- $N_{RS}(OT, OT')$ is a strategy whereby authorizations for objects of type $OT'$ that are obtained by implication from authorizations on objects of type $OT$ are not stored (no redundancy);
- $D_{RS}(OT, OT')$ is a strategy whereby authorizations on objects of type $OT'$ that are obtained by implication from stored authorizations on objects of type $OT$ are stored (redundancy for authorizations that propagate down); and
- $U_{RS}(OT, OT')$ denotes a strategy whereby authorizations on objects of type $OT'$ that are obtained by implication from stored authorizations on objects of type $OT'$ are stored (redundancy for authorizations that propagate up).
Examples of strategies for some authorization types are the following (in the examples we reference the catalogs shown in Figure 21):

— **D_RS(Class, Setof-Instances):** If a subject $S$ has an authorization on a class $C$ (stored in the ClassAB catalog), we also store the implied authorizations on $\text{setof-Instances}(C)$ in the SetofInstancesAB catalog. There is a one-to-one correspondence between authorization objects of the types $\text{Class}$ and $\text{setof-Instances}$ (cf. Figure 15). Therefore the size of the SetofInstancesAB catalog does not increase much and the ClassAB catalog need not be accessed when checking authorization on the $\text{setof-Instances}$ of a given class $C$. The same discussion applies to authorization types $\text{Class}$ and $\text{setof-Comp-Instances}$.

— **U_RS(Setof-Instance-Instance, Instance):** If a subject $S$ has an authorization on an instance of a given class $C$ (stored in the InstanceAB catalog), an authorization for $S$ is also recorded for $\text{setof-Instances}(C)$ in the SetofInstancesAB catalog. The reason for this redundancy is to support instance-level authorization more efficiently. In fact, when checking authorization for a subject $S$ on an instance and when no authorizations are found for $S$ in the SetofInstancesAB catalog, then there is no need to access the catalog InstanceAB.

— **D_RS(Comp-Instance, Instance):** If a subject $S$ has an authorization on a composite instance, all authorizations on component instances are computed and stored as well. In this way, when accessing a component instance directly there is no need to determine its root object(s) for checking the authorization.

5.2.2 **Access Strategies.** Each authorization operation requires access to a number of authorization catalogs. For example, if we are to check authorization for a subject $S$ on an instance, we may first check if the user has authorization on the system. If $S$ has no such authorization, authorization on a database is checked, and so forth.

Suppose that authorization on an object $O$ for a subject $S$ must be checked. Three of the possible strategies are as follows:

1. **Top-down:** The search starts from the catalog associated with the topmost object type (i.e., SystemAB). If no authorization is found for $S$, then the catalog associated with the next lower-level object type in the AOS is accessed. The search continues until (in the worst case) the catalog associated with the object type of $O$ is accessed.

2. **Top-down with early termination:** This strategy is a variation of the top-down strategy, in that the search starts from the catalog associated with the topmost object type. However, if no authorization is found for $S$, then the search stops. This strategy requires redundancy for authorizations that propagate up the granularity hierarchy.

3. **Bottom-up:** The search starts from the catalog associated with the object type of $O$. If no authorization is found for $S$, then the catalog associated with the next-higher-level object type is accessed. The search continues until (in the worst case) the SystemAB catalog is accessed.
Which strategy will be the best depends on how many users have authorizations on particular object types and on which object types the users have total authorizations. We have adopted the simple top-down strategy. This decision is based on the consideration that the strategy performs well when most of the users have total authorizations on classes. In many application environments, we expect that the users will access many instances of classes of interest to them and will tend to have total authorizations on the classes. In other environments, the users will work with a small number of large objects and will not need total authorizations on their classes. Therefore, the performance of the top-down strategy for instance-level authorization check is not significantly worse than the bottom-up strategy. We also note that applying the top-down with early termination approach to all authorization object types would mean recording partial authorizations in the SystemAB and DatabaseAB catalogs for all users having authorizations at any level in the AOS. Therefore, the size of the two catalogs would increase, making it difficult to fully cache them.

We now summarize the access strategies for different authorization object types. We assume that the D_RS(Class, SetofInstances) strategy is applied:

1. **Database**: The SystemAB catalog is accessed first; if no authorization is found or if a weak authorization with exceptions is found, the DatabaseAB catalog is accessed.

2. **Class**
   
   (i) If the access is to a class definition (i.e., read or modify a class definition, or create a subclass), then the SystemAB catalog is accessed first. If the authorization is not found or if a weak authorization with exceptions is found, the DatabaseAB catalog is accessed; and, if again the authorization is not found, the ClassAB catalog is accessed.

   (ii) If the operation is on the instances of a class (i.e., read or modify instances, as in queries), the strategy of (2i) is used, except that the SetofInstancesAB catalog is accessed, instead of the ClassAB catalog. This is because of the redundancy between authorization object types Class and setofInstances. In fact, if an authorization is granted on a class (stored in the ClassAB catalog), the corresponding authorization on the setofInstances of that class is stored in the SetofInstancesAB catalog.

   (iii) If a method is to be executed, the strategy of case (2ii) is applied, except that the Method_AttributeAB catalog is accessed instead of the SetofInstancesAB catalog.

   (iv) If an index is to be created or dropped, the strategy of case (2ii) is applied, except that the IndexAB catalog is accessed instead of the SetofInstancesAB catalog.

3. **Instance**: The strategy of case (2ii) is applied. However, if no authorization is found in the SetofInstancesAB catalog, the InstanceAB catalog is accessed.

From the discussion we can see that, in the worst case, authorization checking on a class requires access to three catalogs. Authorization checking at the instance level requires four catalog accesses in the worst case (i.e., the user does not have a read authorization on the class to which the instances belong). However, if the SystemAB catalog is not implemented (as in the case of a single shared database) and the DatabaseAB catalog is cached in main memory (or if both of them are cached in main memory), only one access to a catalog on secondary storage will be necessary to check authorizations on classes, and at most two accesses to check authorizations at the instance level.

6. SUMMARY

Next-generation database systems are expected to support rich data models that incorporate object-oriented concepts and some of the key semantic data modeling concepts, such as composite objects (part hierarchies) and versions. In this paper we have presented a model of authorization that can be used as the basis for an authorization mechanism in such database systems, and we also have proposed solutions to a number of key issues in implementing the model in a database system. The model we have presented represents a significant extension of the model developed earlier in [14].

We first introduced key concepts that form the basis of the model. The model refines the concept of implicit authorizations first formalized in [5] and [6]; the model extends the concept along each of the three dimensions in authorization definitions, namely, the authorization subject, authorization type, and authorization object. Furthermore, it refines the authorization type into strong and weak authorizations, and distinguishes a positive authorization and a negative authorization.

Next, we developed rules for computing implicit authorizations from an explicitly defined authorization, along each of the three authorization dimensions. We also formalized the semantics of weak authorizations, namely, rules for computing authorizations implied by weak authorizations and the consequences of overriding implied weak authorizations. Then we applied the concept of implicit authorizations on object-oriented and semantic modeling concepts; in particular, the properties of a class, the class hierarchy, composite objects, and versions. The ORION object-oriented database system is one system supporting these modeling concepts [1, 3, 11, 12]. The treatment of authorizations for these modeling concepts represents a reasonable clarification and enhancement of that given in [14].

Finally, we discussed a number of key issues that arise in implementing our model, including storage organization for stored authorizations, and strategies to access the stored authorizations. A significant subset of the model of authorization we presented has been implemented in the ORION system.

There are some important open issues that require further research. These include mandatory security, and authorizations based on database contents and context. A preliminary discussion of these issues for object-oriented
databases is given in [15]. Furthermore, in object-oriented databases, it is
interesting to consider making each method an authorization type, although
there seems to be the practical difficulty of defining authorization implication
orderings among them.

APPENDIX A: EXAMPLE OF CONFLICTING WEAK AUTHORIZATION

We provide an example of conflicting weak authorization by considering the
case of composite objects. Two composite objects and their components are
illustrated in Figure 23a. For example, comp-instance[1] consists of two
components, comp-instance[3] and comp-instance[4]. These two components
are in turn composite objects since they consist of other objects. Note that
comp-instance[4] is shared between two composite objects.

Now consider the weak authorizations \([s, \text{comp-instance}[1], W]\) and
\([s, \text{comp-instance}[2], W]\). Since authorizations are implied along composite
object hierarchies, we see that an implicit weak write authorization is
defined on all components of comp-instance[1]; similarly for the negative
authorization on comp-instance[2]. The scopes of the two weak authorizations
are represented in Figure 23b. Note that the scopes of the two explicit
authorizations overlap on the objects enclosed in the shadowed area. Since
one authorization is the negation of the other, they conflict. This means that,
if the authorization on comp-instance[1] is defined before the authorization on
comp-instance[2], the second will be rejected by the system.

One way to solve this conflict and to allow the entering of the authorization
on comp-instance[2] is for the user to define an explicit weak authorization on
the component rooted at comp-instance[4]. In this way the scope of this
authorization is subtracted from the scope of the authorizations on comp-
instance[1] and comp-instance[2]. Let us assume that the user defines a weak
positive write authorization on comp-instance[4]. This authorization is redun-
dant with the authorization on comp-instance[1]. However, it is useful since it
allows a later definition of a negative write authorization on comp-
instance[2] without conflicts. The scopes of the three authorizations are
shown in Figure 23c. Note that the scopes of authorizations \([s, \text{comp-
instance}[1], W]\) and \([s, \text{comp-instance}[2], W]\) do not overlap.

APPENDIX B: STRONG IMPLICATION RULES

The following rule and corollary summarize the computation of implicit
authorizations along the three authorization domains, \(S\) (authorization sub-
ject), \(O\) (authorization objects), and \(A\) (authorization types). They are ob-
tained by combining Rules 1, 2, and 3 defined in Section 3. All three
authorization domains are organized as lattices, with a partial ordering
denoted by \(\geq\). The notation \(s_i \geq s_j\), where \(s_i, s_j \in S\), indicates that either
\(s_i = s_j\) or there exist \(s_1, \ldots, s_n\) such that \(s_i \geq s_1 \geq \cdots \geq s_n\). Furthermore,
the symbol \(\rightarrow\) denotes implication between authorizations.

**Rule I.** For any \(s_i \in S\) and \(s_k \in S\), \(o_i \in O\) and \(o_j \in O\), \(a_n \in A.\downarrow\), and
\(a_m \in A\), if \(s_k \geq s_i, o_i \geq o_j\), and \(a_n \geq a_m\) then
if \(c(o_j, a_m) = T\) then \((s_i, o_i, a_n) \rightarrow (s_k, o_j, a_m)\).
Fig. 23. (a) Composite object granularity hierarchy; (b) scopes of authorizations on comp-instance[1] and comp-instance[2]; (c) scopes of authorizations on comp-instance[1], comp-instance[2], and comp-instance[4].
For any $s_i \in S$ and $s_k \in S$, $o_i \in O$ and $o_j \in O$, $a_n \in A$, and $a_m \in A.up$, if $s_k \geq s_i$, $o_i \geq o_j$, and $a_n \geq a_m$ then

if $c(o_i, a_m) = T$ then $(s_i, o_j, a_n) \rightarrow (s_k, o_i, a_m)$.

For any $s_i \in S$ and $s_k \in S$, $o_i \in O$ and $o_j \in O$, $a_n \in A.nil$, and $a_m \in A$, if $s_k \geq s_i$, $o_i = o_j$, and $a_n \geq a_m$ then

if $c(o_i, a_m) = T$ then $(s_i, o_i, a_n) \rightarrow (s_k, o_i, a_m)$.

APPENDIX C: WEAK IMPLICATION RULES

To define weak authorization implications, we need to define the set $P[sl, o_i, a_n]$, that is, the scope of each weak authorization $[s_i, o_i, a_n]$ in the WAB. This is the set of weak authorizations that can be deduced from $[s_i, o_i, a_n]$ using the rules for (strong) authorization implication (see Section 3). The definition of $P[sl, o_i, a_n]$ must satisfy the two invariants of the WAB and the coexistence invariant of the WAB and AB.

Rule WI-P. For any weak authorization definition $[s_i, o_i, a_n] \in WAB$, with $a$ positive or negative, the set $P[sl, o_i, a_n]$ consists of all the weak authorizations $[s, o, a]$, with $s \in S$, $o \in O$, $a \in A$, with $a$ positive and negative, such that

$$(s_i, o_i, a_n) \rightarrow (s, o, a) \text{ and there exists no } [s_k, o_j, a_m] \text{ in WAB, such that }$$

$$(s_i, o_i, a_n) \rightarrow (s_k, o_j, a_m) \text{ and } (s_k, o_j, a_m) \rightarrow (s, o, a).$$

The scope of $[s_i, o_i, a_n]$ ends when the scope of another more specific explicit authorization starts $([s_k, o_j, a_m]$ in the previous definition). Then, we can derive the rule for weak authorization implication.

Rule WI. For any $s_i \in S$ and $s_k \in S$, $o_i \in O$ and $o_j \in O$, $a_n \in A$ and $a_m \in A$, with $a_n$ and $a_m$ positive or negative, if $[s_i, o_i, a_n] \in WAB$ and $[s_k, o_j, a_m] \in P[s_l, o_i, a_n]$ then $[s_i, o_i, a_n] \rightarrow [s_k, o_j, a_m]$.

If we consider the implications only on the domains $O$ and $A$, the weak authorizations that are really meaningful in the WAB are those with positive $a \in A.down$ and negative $a \in A.up$, that is, $W$, $R$, $\neg WA$, and $\neg RD$. In fact, consider the weak authorization $[s, o, a]$, where $a$ is either a positive $a \in A.up$.
or a negative $a \in A. down$. The scope of $[s, o, a]$ would be limited to the set of objects $o'_i$ such that $o'_i \geq o$, that is, the set of objects in a sequence of ILs from the topmost authorization object (i.e., system) to $o$. This scope is practically useless because any strong or weak authorization defined in it, and that would constitute an exception to $[s, o, a]$, would cause the automatic deletion of $[s, o, a]$ from the WAB (because of the invariants requiring consistency of the WAB and coexistence of the WAB and the AB).

Furthermore, since the function $d$ must be fully defined in the scope of $[s_i, o_i, a_i]$ by inference from $[s, o, a]$ using authorization implications, a weak authorization with a weak $R$, in the WAB, should be coupled with a weak authorization with a weak $\neg WA$. This implies a scope of implicit weak authorizations with read authorization type, but without write. Thus, considering the implication of weak authorizations on the domains $O$ and $A$, only three combinations are really meaningful: $W, R$ (coupled with $\neg WA$), and $\neg RD$. Obviously, if we also consider the domain $S$, it is meaningful to define in the WAB explicit weak authorizations with $a \in A.nil$ or positive $a \in A.up$ or negative $a \in A.down$.

For the completeness invariant of the WAB, we require that for each role (unless it can be derived from a subrole) there must be a weak authorization on the root object $o_0$ (in the AOS, it is the system). If nothing else is defined for subject $S$, we assume a weak authorization $[s, o_0, \neg RD]$ in the WAB, which corresponds to the system closure.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous referees and Gio Wiederhold for the detailed comments that have helped us to improve the contents and the presentation of this paper.

REFERENCES


Received December 1988; revised July 1989; accepted January 1990.