A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems

Suh-Yin Lee, *Member, IEEE Computer Society*, and Ruey-Long Liou

Abstract—A locking model adopting a multi-granularity approach is proposed for concurrency control in object-oriented database systems. The model is motivated by a desire to provide high concurrency and low locking overhead in accessing objects. Locking in schemas and locking in instances are developed separately and then are integrated. Schema changes and composite objects are also taken into account. A dual queue scheme for efficient scheduling of lock requests is developed. The model consists of a rich set of lock modes, a compatibility matrix, and a locking protocol. Characteristic query examples on single class, class lattice, and composite objects are used to illustrate the comparison between the ORION model and the proposed model. It is shown that our locking model has indeed made some improvements and is suitable for concurrency control in object-oriented databases.

Index Terms—Object-oriented database, locking model, concurrency control, locking granularity, compatibility matrix.

1 Introduction

Recently, there has been much interest in object-oriented database systems (OODB), which appears to be driven mainly by data-intensive application demands such as CAD/CAM, office information systems and software development environments. With its flexible data model and object-oriented programming paradigm, it is believed that OODB has great potential to be applied widely.

The problems of concurrency control in conventional databases, such as the lost update problem and the uncommitted dependency problem [3], [7], remain in OODB. Furthermore, due to the complexity of the object-oriented data model, the problems become more complicated. One of the main techniques used to control concurrency is based on the concept of locks. The object-oriented data model also complicates the lock requirements [13], [12], [4]. The locking model implemented in ORION still has the following undesirable features:

1) The degree of concurrency is low on locking composite objects. By the locking model, if one transaction updates any component object of a composite object, the composite object is locked in its entirety. No other transactions can directly access any component object of the composite object via the component classes. The degree of concurrency, therefore, is not satisfactory for applications with long transactions.
2) The model does not support schema locks. The model always locks schemas and instances at the same time. Actually, schema reading can be independent of instance accessing for different requests. Concurrency can be increased if individual lock modes for schema accessing are provided.

The term "granularity" refers to the size of the objects that can be locked. The advantage of a coarse granularity is that there are fewer locks and hence less overhead in testing, setting and maintaining those locks. The disadvantage, of course, is that there will be less concurrency. Since different transactions obiously have different characteristics and requirements, it is desirable that the DBMS provides a range of locking granularities (Indeed, many DBMSs do).

The objective of this paper is to develop a feasible locking model for an object-oriented database system which overcomes the shortcomings while retaining the advantages of other locking models. Our model is based mainly on the concept of multi-granularity locking. It contains a rich set of lock modes with different lockable granules, a compatibility matrix, and a complete protocol. Locking for composite objects and schema evolution are presented. We also discuss how the locking requests are scheduled and granted. A dual queue scheduling scheme is proposed to enhance concurrency. These features make our model suitable for OODB.

2 Background

Although there is some common consensus [1], [2], [22], [23], [24], [25], [26], [27] on what object-oriention means and what an object-oriented database is, each object-oriented database system has its own data model. We will first introduce a basic object-oriented data model. Then, some specific features relevant to our proposed locking model are reviewed.

• *S.-Y. Lee is with the Institute of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 300, Republic of China. E-mail: sylee@csie.nctu.edu.tw.*
• *R.-L. Liou is with the Institute for Information Industry, Taipei, Taiwan, Republic of China. E-mail: rliou@iiidns.iii.org.tw.*

Manuscript received Apr. 15, 1992; revised June 1, 1994.

For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number K96012.

2.1 Fundamentals of Object-Oriented Data Model

Though it is not clear how to define a formal model that includes all object-oriented concepts, but some essential concepts are common and are described as follows:

- *Object:* In object-oriented systems, any real-world entity of interest is uniformly modelled as an object. An object can represent not only a simple integer or string but also a complicated vehicle, or an electrical device.
- *Attribute:* Users can define a set of attributes to capture the state of an object. The value for an attribute of an object is itself an object in its own right.
- *Method:* O-O system also provides the ability for the user to define a set of operations, called methods, to capture the behavior of an object. Methods consist of program codes that manipulate or return the state of an object.
- *Message:* State and behavior are encapsulated in an object. Methods as well as attributes are invisible from outside of the object. Objects communicate and perform all operations via message passing. Methods are to procedures or functions what messages are to function calls.
- *Class and Instance:* A class is specified as means of grouping of all the objects that share the same set of attributes and methods. Objects that belong to a class are called instances of that class. A class is similar to an abstract data type. A primitive class is one which has no attributes, such as integer, string, and Boolean.
- *Class Hierarchy and Inheritance:* Users can derive a new class from an existing class. The new class, called a subclass of the existing class, inherits all the attributes and methods of the existing class. The existing class is a superclass of the new class. Classes therefore are arranged in a class hierarchy from the most general to the most specific, in which an edge represents the IS-A relationship.
- *Multiple Inheritance:* A class may inherit attributes and methods from more than one superclasses. The class hierarchy is relaxed to a directed acyclic graph (DAG), or a class lattice.

2.2 Schema Evolution

Schema evolution allows users to dynamically make a wide variety of changes to the database definitions. Existing databases allow only few types of schema changes. For example, SQL/DB allows only dynamic creation and deletion of relations and addition of new columns [20]. The applications supported are record-oriented and do not require a rich set of schema change operations.

The database schema for an object-oriented database has two dimensions. One dimension (vertical) is the class hierarchy which captures the generalization relationship between a class and its subclasses. Another dimension (horizontal) is the class composition hierarchy which represents the aggregation relationship between a class and its attributes and the domains of the attributes.

Two types of changes to the database schema are often necessary [11], [18]. Changes to the class definitions and changes to the structure of a class lattice. Changes to the class definitions include adding and deleting attributes and methods. Changes to the structure include creation and deletion of a class, and alteration of the IS-A relationship between classes. A formal framework for the schema evolution of ORION was presented in [11]. The semantics of class hierarchy is what complicates schema changes and locking requirements. For example, when a class is dropped, all its subclasses will lose the attributes and methods that they had inherited from the class.

2.3 Composite Objects

There are two major relationships among objects in an object-oriented system. IS-A relationship supports the class inheritance. The IS-PART-OF relationship captures the notion of composition that an object is a part of another object. Many applications require the capability to define, store and retrieve a collection of related objects as a single logical object for the purpose of semantic integrity, and efficient storage and retrieval [10]. For example, an aircraft composed of a fuselage, wings, engines and a landing gear can be viewed as a composite object.

A composite object has a special object, called the composite root object. The root of a composite object contains a number of component objects (or dependent objects). Each component object can be a simple object (with no component objects) or it may contain its own component objects. In a composite object, no component object can be referenced more than once. Thus a composite object is a hierarchy of objects. The classes to which the objects of a composite object belong are also organized in a hierarchy, called a composite object schema.

Fig. 1 depicts a composite object schema and two instances of the composite object hierarchy. The set of classes Vehicle, AutoBody, AutoDrivetrain, forms a composite object schema. The set of instances V1, B1, and D1 forms a composite object hierarchy and V1 is the root object. The set of instances V2, B2, and D2 forms another composite object hierarchy and V2 is the root object.

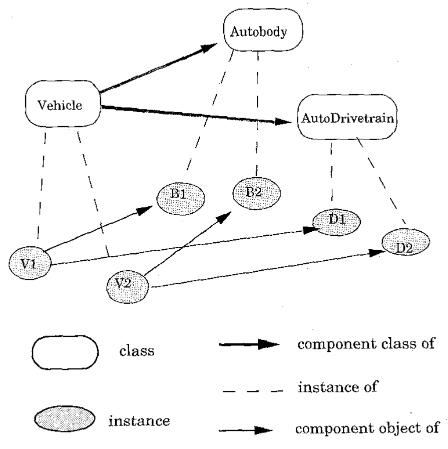


Fig. 1. Composite object hierarchy.

2.4 Extended Composite Objects

Although the feature of composite objects has been found quite useful, the model presented in the previous subsection suffers from a number of serious shortcomings.

1) The model restricts a composite object to a strict hierarchy of exclusive component objects. That is the component object can only be the part of single one composite object. This restriction is not suitable in many applications. For example, an identical chapter may be contained in two different books.
2) The model forces a top-down creation of a composite object; that is, before a component object may be created its parent object must already exist. This prevents a bottom-up creation of objects by assembling already existing objects.
3) The model requires that the existence of a component object depends on the existence of the parent object. If a composite object is deleted, all its component objects are also deleted. This dependence impedes the reuse of part objects in a complex design environment.

In order to eliminate these shortcomings while retaining the advantages of the previous model, semantic extensions are necessary. Kim, Bertino, and Garza [15] describes significant semantic extensions to the model by extending the dependent exclusive composite reference to four references. The semantics of a composite reference is refined on the basis of whether an object is a part of only one object (exclusive) or more than one object (shared). Another refinement is based on whether the existence of an object depends on the existence of its parent object, i.e., a composite reference may be dependent or independent. The extension reveals the following four types of composite reference:

1) dependent exclusive composite reference
2) independent exclusive composite reference
3) dependent shared composite reference
4) independent shared composite reference

We may change the reference type of Body attribute in Fig. 1 from dependent exclusive type to dependent shared type. Then, there may be another instance V3 of vehicle that has shared reference to the same component object, say B2, with V2. The composite object hierarchy is relaxed to a lattice.

3 Concurrency Control in OODB

3.1 Conflict Problems in OODB

In conventional database systems, the general concurrency problems include the lost update problem, the uncommitted dependency problem, and the inconsistency problem [7]. One of the main techniques used to control concurrency is based on locking. These concurrency control problems may exist on schemas and instances in OODB. We classify the access conflicts into three kinds:

1) *instance-instance conflict*: An instance-instance conflict occurs when more than one transaction accesses the same instance concurrently. An instance in OODB may be a primitive object or a composite object. For two transactions accessing two different composite objects with some common, shared component objects, conflicts may occur on these component objects. This kind of conflict will not be detected until the IS-PART-OF relationship between objects is verified. Concurrency control involving composite objects needs special handling.
2) *instance-schema conflict*: An instance-schema conflict occurs if some transaction is accessing instances while another is accessing the schema of the instances. Schemas keep the definitions of instances. The inconsistency problem is likely to occur when the schemas of the accessed instances are modified. Concurrency control in OODB that allows schema evolution should take instance-schema conflicts into account.
3) *schema-schema conflict*: A schema-schema conflict occurs if more than one transaction is accessing "associated" schemas concurrently. Schemas are associated by the inheritance mechanism in the class hierarchy. Any change in a superclass schema is inherited by all its subclasses. Concurrency control therefore becomes difficult due to the inheritance mechanism of schemas.

3.2 Locking Model in ORION

Not much systematic and in depth discussion on concurrency control problems in OODB can be found in the literature except in ORION system [10], [4], [12],[19]. In the following, we will describe and use the ORION model for illustration and comparison [13], [14], [15]. The locking model in ORION is based on Gray's hierarchy locking model [8], [9]. A two-level hierarchy as depicted in Fig. 2 is used to model the lockable granules in OODB. The coarse granule is the class level and the fine granule is the instance level.

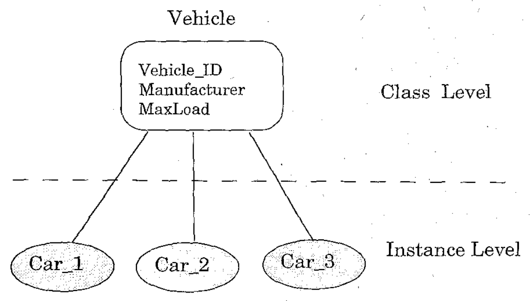


Fig. 2. Two-level lock hierarchy.

Instance objects in ORION system are locked only in S or X mode. Class objects can be locked in S, X, IS, IX, or SIX mode. The semantics of these modes are stated as follows:

1) Instance Objects:

S:	A shared lock on an instance means that the instance can be read.
X:	An exclusive lock on an instance means that the instance can be read or updated.

2) Class Objects:

S: A shared lock on a class means that the class definition is to be locked in S mode, and all instances of the class are implicitly locked in S mode.

X: An exclusive lock on a class means that the class definition is to be locked in X mode, and all instances of the class are implicitly locked in X mode.

IS: An intention shared lock on a class means that the instances of the class are to be explicitly locked in S mode as required.

IX: An intention exclusive lock on a class means that the instances of the class are to be explicitly locked in S or X mode as necessary.

SIX: A shared intention exclusive lock on a class means that the class definition is locked in S mode, and all instances of the class are implicitly locked in S mode. Also the instances to be updated will be explicitly locked in X mode.

Fig. 2 shows the lock hierarchy of Vehicle class with some instances. The following query examples on Vehicle class illustrate the use of the lock modes and the locking protocol.

1) Select all Vehicle instances that are manufactured by Ford.
   a) lock Vehicle class in IS mode
   b) lock the selected Vehicle instances in S mode
2) Update instances of Vehicle that have maxload of 2 tons.
   a) lock Vehicle class in IX mode
   b) lock the selected Vehicle instances in X mode

In order to recognize a composite object as a single lockable granule three new lock modes ISO, IXO, SIXO are added corresponding to the IS, IX, SIX modes, respectively [13]. To lock a composite object, the root class is locked in IS, IX, S, SIX, or X mode as before. Each of the component classes in the composite object schema is locked in ISO, IXO, S, SIXO, X mode, respectively. The three lock modes added are dedicated to component classes and can prevent a transaction from accessing the component objects of a composite object O while another transaction is accessing the entire composite object O.

The following query examples on the composite class Vehicle illustrate the use of the lock modes and the locking protocol for accessing composite objects.

1) Select instances of Vehicle that ...
   a) lock Vehicle class in IS mode
   b) lock selected Vehicle instances in S mode
   c) lock the component classes in ISO mode
2) Update instances of Vehicle that ...
   a) lock Vehicle class in IX mode
   b) lock selected Vehicle instances in X mode
   c) lock the component classes in IXO mode

The locking model described above, which is based on object data model of [12], is applicable to composite objects consisting of exclusive composite references only. It is again extended for the shared composite references.

A component class of exclusive reference is locked as before. However, three new lock modes are introduced for the component class of shared references: ISOS (intention shared object-shared), IXOS (intention exclusive object-shared), SIXOS (shared intention exclusive object-shared), which correspond to the ISO, IXO, and SIXO, respectively. Table 1 shows the compatibility matrix including the extended set of lock modes [15].

TABLE 1
COMPATIBILITY MATRIX FOR GRANULARITY LOCKING AND SHARED/EXCLUSIVE COMPOSITE OBJECT LOCKING [15]

Current Mode	Requested Mode										
	IS	IX	S	SIX	X	ISO	IXO	SIXO	ISOS	IXOS	SIXOS
IS	Y	Y	Y	Y	N	Y	N	N	Y	N	N
IX	Y	Y	N	N	N	N	N	N	N	N	N
S	Y	N	Y	N	N	Y	N	N	Y	N	N
SIX	Y	N	N	N	N	N	N	N	N	N	N
X	N	N	N	N	N	N	N	N	N	N	N
ISO	Y	N	Y	N	N	Y	Y	Y	Y	Y	Y
IXO	N	N	N	N	N	Y	Y	N	Y	Y	N
SIXO	N	N	N	N	N	Y	N	N	Y	N	N
ISOS	Y	N	Y	N	N	Y	Y	Y	Y	N	N
IXOS	N	N	N	N	N	Y	Y	N	N	N	N
SIXOS	N	N	N	N	N	Y	N	N	N	N	N

Let us consider the composite objects in Fig. 3. Component objects Instance[c] and Instance[c′] belong to class C. Instance[w] and Instance[w′] belong to class W. Instance[i], Instance[j], Instance[k] belong to classes I, J, K, respectively. Instance[j] and Instance[k] have shared references to Instance[c′]. The following examples illustrate the extended locking protocol.

1) Update the composite object rooted at Instance[i]
   a) lock class I in IX mode
   b) lock composite object Instance[i] in X mode
   c) lock class C in IXO mode
2) Update the composite object rooted at Instance[j]
   a) lock class J in IX mode
   b) lock composite object Instance[j] in X mode
   c) lock class C in IXOS mode
   d) lock class W in IXO mode

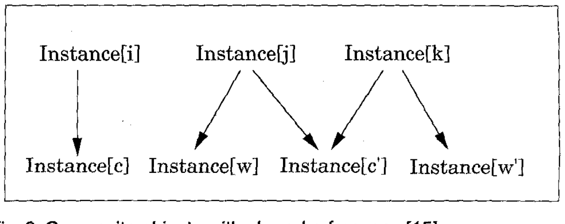


Fig. 3. Composite objects with shared references [15].

4 MULTI-GRANULARITY LOCKING APPROACH

Objects in object-oriented databases may be related by semantic relationships such as IS-A, IS-PART-OF, or by version derivation relationships. Sometimes we intend to lock a single instance (simple object), in other times we may intend to lock a group of related objects. Transactions may last in varying duration and may have different characteristics and requirements. Since the targets to be locked may vary depending on the applications in OODB, it is desirable and advantageous to provide a range of different lock granules [5], [6], [8], [16], [17]. The aim is to achieve high degree of concurrency and low overhead. This paper presents a multi-granularity locking model (MGL)[21] which consists of three components:

- a set of lock modes on different granules
- a compatibility matrix to indicate whether two lock modes are compatible
- a complete protocol to guide how lock requests are issued.

The MGL model for OODB is quite different from the locking model for conventional databases. It has to take some features of OODB such as class hierarchy, composite object hierarchy and schema evolution into account. In MGL model, schema locking and instance locking are developed and then are integrated. For instance locking, composite objects are distinguished from primitive objects and some lock modes that treat the composite objects as logical lockable granules are provided. The detailed proof of the protocols that guide the correct issuing of locking requests can be found in [21]. In the proof, we suppose all transactions obey the MGL protocol with respect to a given class hierarchy (lattice). If a transaction owns an explicit or implicit lock on an instance which belongs to some class in the hierarchy, then no other transaction can own a conflicting explicit or implicit lock on that instance.

4.1 Locking in Schemas

The classes in OODB are organized as a specialization/generalization hierarchy. A class inherits all attributes and methods from each of its superclasses. The schema of a class C is composed of the inherited schema and the local defined schema. To read the schema of a class C, the schema of all superclasses of class C are also read, and has nothing to do with the subclasses of class C. That is, to read the schema of class C, class C as well as all the superclasses of class C are locked in shared mode. No locks should be set on any subclasses of class C.

Any change in the definition of class C also needs to read the definitions of all superclasses of class C. Because these changes on class C will be inherited by all subclasses of class C, the schema in all subclasses of class C are also changed. To write schema of class C, all superclasses of C have to be locked in shared mode. Class C and all its subclasses are locked in exclusive mode.

From the discussion above, two schema locks, RS and WS modes, are proposed to satisfy these requirements on locking schemas.

- An RS lock on class C means the schema (definition) of class C is locked in shared mode. Other transactions can read the schema or read/write instances of class C.
- A WS lock on class C means the schema (definition) of class C is locked in exclusive mode explicitly. Definitions of subclasses of class C are locked in exclusive mode implicitly. No other transactions are allowed to access either schemas or instances of the lattice rooted at class C.

The compatibility matrix for schema locking is shown in Table 2. The protocol for locking schema is as follows:

Schema Locking Protocol A:

a) Before requesting an RS or WS lock on a class, one should request at least one of its superclasses in RS lock.
b) If a class which has more than one superclass is locked in WS mode implicitly, change the implicit WS lock to explicit WS lock.
c) Before changing an edge (the relationship between two classes), the classes connected by the edge must be set WS lock implicitly or explicitly.
d) Set all locks in root-to-leaf order.
e) Locks should be released either at the end of a transaction (in any order) or in leaf-to-root order before the end of a transaction.

TABLE 2
COMPATIBILITY MATRIX FOR SCHEMA LOCKING

		Requested Mode	
		RS	WS
Current Mode	RS	Y	N
	WS	N	N

In Protocol A, (a) shows that RS mode serves as an intention mode because WS mode implies implicit locking on the schema of the subclasses of a class. (b) is required for detecting conflicts on classes with multiple inheritance. The class lattice may be changed by modifying edges. (c) insures that these changes are correct. To prevent deadlock, all lock requests must follow some order. The order of requesting and releasing locks are defined in (d) and (e).

4.2 Locking in Instances

The fundamental motivation of the multiple granularity locking protocol is to minimize the number of locks to be set in accessing the database [20], [9]. We follow this approach on locking instances. First of all, we identify the lockable granules.

The arrangement of instances in OODB systems resembles a hierarchical structure with similar objects grouped into a class and similar classes generalized into a superclass. A locking graph for OODB is as shown in Fig. 4. Three lattices are shown, class lattice, class-instance hierarchy and composite object lattice. A simple object is a special case of a composite object.

For the class-instance hierarchy, only two lockable granules are allowed, class granule and instance granule. If most of the instances of a class are to be accessed, it makes sense to set one lock on the class granule, rather than one lock for each instance. Thus a lock on a class will imply a lock on each instance of the class. The locking overhead is low. When few instances of a class need to be accessed, it is better to lock the instances individually to improve concurrency.

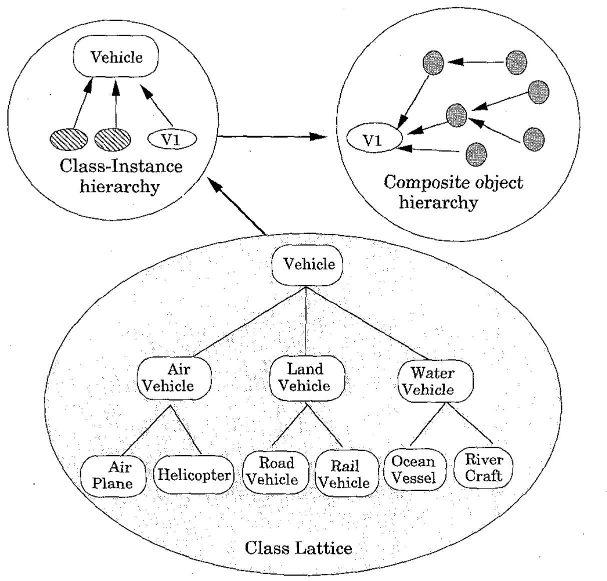


Fig. 4. Three hierarchies of data in OODB.

For the class-instance hierarchy, five modes which are similar to the single-class locking model of the ORION system are provided. Their semantic meanings are as follows:

S: A shared lock on a class means that all instances of the class are implicitly locked in S mode.

X: An exclusive lock on a class means that all instances of the class are implicitly locked in X mode.

IS: An intention shared lock on a class means that instances of the class are to be explicitly locked in S mode as necessary.

IX: An intention exclusive lock on a class means that instances of the class are to be explicitly locked in S or X mode as necessary.

SIX: A shared intention exclusive Lock on a class means that all instances of the class are implicitly locked in S mode and can be explicitly locked in X mode as necessary.

Locking on a class lattice is appropriate for a query against a class when the class or the domains of the attributes of the class are close to the root of a deep class lattice. When most of the instances of a class lattice are to be accessed, it makes sense to set one lock on the class lattice in its entirety, rather than one lock for each class in the lattice. Thus, a lock on a class lattice will imply a lock on each class of the lattice. The locking overhead is low. When few subclasses of a class lattice need to be accessed, it is better to lock the subclass individually to improve concurrency.

Five class-lattice lock modes by making extensions to the lock modes for class-instance hierarchy are proposed. The semantic meanings of the "lattice version" lock modes are listed below.

S*: A shared star lock on a class C means that all classes and instances of the class lattice rooted at class C are implicitly locked in S mode.

X*: An exclusive star lock on a class C means that all classes and instances of the class lattice rooted at class C are implicitly locked in X mode.

IS*: An intention shared star lock on a class C means that all classes of the class lattice rooted at class C are implicitly locked in IS mode. Thus all instances of the lattice are to be explicitly locked in S mode as necessary.

IX*: An intention exclusive star lock on a class C means that the classes of the class lattice rooted at class C are implicitly locked in IX mode. Thus all instances of the lattice are to be explicitly locked in X, or S mode as necessary.

SIX*: A shared intention exclusive star lock on a class C means that all classes of the class lattice rooted at class C are implicitly locked in SIX mode. Thus all instances of the lattice are implicitly locked in S mode and will be explicitly locked in X mode as necessary.

Because the class-lattice locking modes cause implicit locking on each subclass of a class, as depicted in Fig. 5 we need to lock the superclasses of the class in intention modes. Four intention modes: IR, IW, IRI, and IWI are needed. These intention modes are used to tag all superclasses of a class to be locked and prevent other incompatible locks on the superclasses. Their semantic meanings are as follows:

IR: An intention read lock on a class C means that the subclasses of class C are to be locked in S, or S* mode as necessary.

IW: An intention write lock on a class C means that the subclasses of class C are to be locked in X, X*, SIX, or SIX* mode as necessary.

IRI: An intention read instance lock on a class C means that the subclasses of class C are to be locked in IS, or IS* mode as necessary.

IWI: An intention write instance lock on a class C means that the subclasses of class C are to be locked in IX, or IX* mode as necessary.

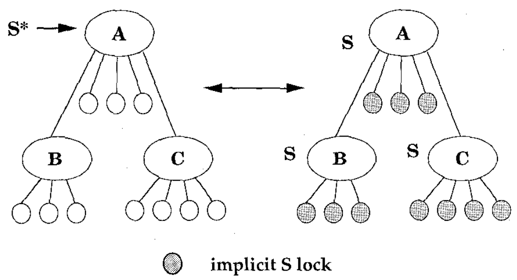


Fig. 5. The semantics of class-lattice locking.

Locking on simple instances and composite objects are quite different. For simple instances two lock modes, S and X modes are enough. For locking on composite objects, the semantic information has to be taken into account. A composite object contains a group of component objects which are also organized as a hierarchy or a lattice. For large composite objects, if most component objects are accessed, setting one lock on the "owner" object rather than setting locks for each component object works better. If only few component objects are accessed, setting locks for accessed component objects can gain more concurrency. Gray's [9] proposed granularity locking can be applied to composite objects. Five modes S, X, IS, IX, SIX are provided in [13] for composite object locking and are described as follows:

S: A shared lock on an instance means that the instance can be read. A shared lock on a composite instance means that the instance is locked in S mode explicitly and all of its component instances are locked in S mode implicitly.

X: An exclusive lock on an instance means that the instance can be read or updated. An exclusive lock on a composite instance means that the instance is locked in X mode explicitly and all of its component instances are locked in X mode implicitly.

IS: An intention shared lock on a composite instance means that the component instances of the instance are to be explicitly locked in S mode as necessary.

IX: An intention exclusive lock on a composite instance means that the component instances of the instance are to be explicitly locked in X or S mode as necessary.

SIX: A shared intention exclusive Lock on a composite instance means that all component instances of the instance are implicitly locked in S mode and can be explicitly locked in X mode as necessary.

The shortcoming of Gray's [9] locking protocol is that it does not recognize a composite object as a single lockable granule. Our protocol modifies Gray's protocol to "understand" composite objects. The protocols for locking on class granules and instance granules are different. The locking protocol for instances are given as follows:

Instance Locking Protocol B:

a) To read an instance, the instance must be held in S or X mode explicitly or implicitly. To write an instance, the instance must be held in X mode explicitly or implicitly.
b) Before requesting an S lock on an instance of class C, class C must be held in IS (IS*) mode or some superclass of class C must be held in IS* mode.
c) Before requesting an X lock on an instance of class C, class C must be held in IX (IX*, SIX, SIX*) mode or one of its superclasses held in IX* (SIX*) mode.
d) Before requesting an S, or IS lock on a composite object, at least one of its parent instance must be held in IS mode. As a consequence, none of its parents along some path to root object can be granted to other incompatible transactions.
e) Before requesting an X, IX or SIX lock on a composite object, at least one of its parent instances must be held in IX mode. As a consequence, no parent instances can be held in a mode incompatible with IX.
f) For composite instances, set all locks in parent-to-child order in the composite hierarchy.
g) For composite instances, locks should be released either at the end of the transaction in any order or in child-to-parent order before the end of the transaction.
h) Before requesting X or S on a composite object, the component objects which are shared referenced must be set X or S lock explicitly.

In Protocol B, (a) enforces that a proper lock must be requested before accessing an instance. (b)(c) tell that a proper lock on classes must be requested in advance before locking instances. How composite objects are locked is defined in (d)(e)(f). For the composite objects with shared references, (h) is needed to check the conflicts on the shared component objects. Our model is also applicable to the ORION's extended composite object data model [15]. To prevent deadlock, all lock requests must follow some order. The order of requesting and releasing locks are defined in (f) and (g).

Before accessing instances, the above protocol requires that classes must hold proper locks to prevent conflicts. The locking protocol on class granules is as follows:

Class Locking Protocol C:

a) Before requesting an S, S*, or IR lock on a class C, at least one superclass of class C must be held in IR or S* mode.
b) Before requesting an X, X*, SIX, SIX*, or IW lock on a class C, at least one superclass of class C must be held in IW, X*, or SIX mode.
c) Before requesting an IS, IS*, or IRI lock on a class C, at least one superclass of class C must be held in IRI or IS* mode.
d) Before requesting an IX, IX*, or IWI lock on a class C, at least one superclass of class C must be held in IWI or IX* mode.
e) If a class C, which has multiple superclasses, is locked in S, X, IS, IX, SIX mode implicitly, then lock S*, X*, IS*, IX*, SIX* is set explicitly on class C, respectively.
f) To request S(S*), X(X*), IS(IS*), IX(IX*), or SIX(SIX*) mode on a composite class, all its component classes in the composite object schema should be locked in S*, X*, IS*, IX*, SIX* mode, respectively.
g) Set all locks in root-to-leaf order.
h) Locks should be released either at the end of the transaction in any order or in leaf-to-root before the end of the transaction.

In Protocol C, (a)(b)(c)(d) tell how intention locks are requested. (e) prevents conflicts occurring on subclasses with multiple inheritance. (f) can recognize a composite object as a single lockable granule. The order of requesting and releasing locks are ruled by (g)(h).

Fourteen lock modes are provided on locking instances. There will be 14*14 elements in the compatibility matrix. Each element indicates whether the two corresponding

TABLE 3
COMPATIBILITY MATRIX FOR INSTANCE LOCKING

Current Mode / Requested Mode

	IS	IX	S	SIX	X	IS*	IX*	S*	SIX*	X*	IR	IW	IRI	IWI
IS	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	Y	Y
IX	Y	Y	N	N	N	Y	Y	N	N	N	Y	Y	Y	Y
S	Y	N	Y	N	N	Y	N	Y	N	N	Y	Y	Y	Y
SIX	Y	N	N	N	N	Y	N	N	N	N	Y	Y	Y	Y
X	N	N	N	N	N	N	N	N	N	N	Y	Y	Y	Y
IS*	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	N	Y	Y
IX*	Y	Y	N	N	N	Y	Y	N	N	N	N	N	Y	Y
S*	Y	N	Y	N	N	Y	N	Y	N	N	Y	N	Y	N
SIX*	Y	N	N	N	N	Y	N	N	N	N	N	N	Y	N
X*	N	N	N	N	N	N	N	N	N	N	N	N	N	N
IR	Y	Y	Y	Y	Y	Y	N	Y	N	N	Y	Y	Y	Y
IW	Y	Y	Y	Y	Y	N	N	N	N	N	Y	Y	Y	Y
IRI	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y
IWI	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y

locks can be set on the same object concurrently. Let us take a look at these lock modes again. Each of the lock modes captures two semantic meanings: access mode and effective scope of locking. For example, in "set X* on class C," the access type of X* is write type and effective locking scope is all instances belonging to the lattice rooted at class C. Two lock modes are compatible if their access types are concurrently executable, e.g., two read accesses. If the access modes are conflicting, we may check further whether their effective locking scopes are overlapped. If not overlapped, concurrent execution is still possible. Based on this analysis, we derive a simple procedure to generate each element of the compatibility matrix shown in Table 3.

The following query examples illustrate protocols B and C on locking instances:

1) Select all instances of class Vehicle and all its subclasses that use gasoline as fuel (Fig. 6). Suppose that all the instances are simple instances.
   a) lock Vehicle class in IS* mode
   b) lock the selected instances in classes Vehicle, LandVehicle, RoadVehicle and RailVehicle in S mode
2) Update instances of class AirVehicle that use propellers (Fig. 6). Suppose that the instances are all simple instances.
   a) lock Vehicle class in IWI mode
   b) lock AirVehicle class in IX mode
   c) lock the selected AirVehicle instances in X mode
3) Update instances B2 of class AutoBody (Fig. 1).
   a) lock superclasses of class AutoBody in IWI mode
   b) lock class AutoBody in IX mode
   c) lock all component classes of AutoBody in IX* mode
   d) lock the parent object (V2) of instance B2 in IX mode
   e) lock instance B2 in X mode

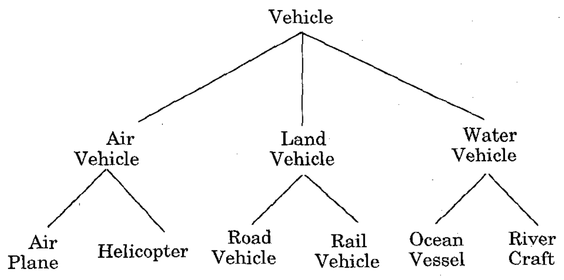


Fig. 6. A class hierarchy: Vehicle.

4.3 Integration of Schema Locking and Instance Locking

Although the problem of propagating the changes of schemas to instances is not defactor, concurrency among instances and schemas access may exist in OODB, especially in the design environments. The schema locking and instance locking mechanism developed above must be integrated to be a unified model. The three components of the MGL model: lock modes, protocols and compatibility matrices must be integrated. Although protocols A, B, C have distinct lock modes, they follow the same procedure in lock setting and lock releasing. Both lock modes and protocols can be combined together without any modification.

As mentioned before, schema reading is independent of instance access. RS mode thus is compatible with every instance locking mode. Every instance lock also has to lock associated schemas in RS mode, which is incompatible with WS mode. This leads to the derivation that every instance lock mode conflicts with WS mode. Table 4 shows the integrated compatibility matrix. The block with dots is the compatibility matrix in Table 3.

TABLE 4
INTEGRATED COMPATIBILITY MATRIX

		Requested Mode		
		RS	WS	Li
Current Mode	RS	Y	N	Y
	WS	N	N	N
	Li	Y	N	▨

Li: instance lock mode

4.4 Locking Scheduling

Any lockable object in OODB can be viewed as a resource in a computer system. The set of all requests for a particular object is kept in a queue sorted by some fair scheduler as indicated in Fig. 7. A fair scheduler must guarantee that no particular transaction will be blocked indefinitely [9]. In other words, a transaction which is forced to wait due to the locking protocol will eventually come out of the wait state.

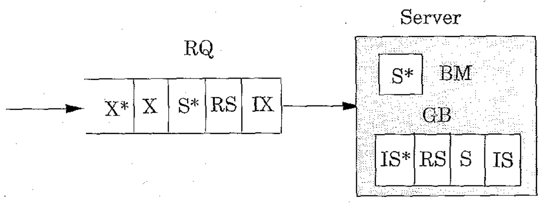


Fig. 7. Diagram of lock requests scheduling: FCFS.

The granted requests are kept in a buffer called granted buffer(GB). All requests in the granted buffer are mutually compatible and are granted concurrently. The combining mode of all the requested modes in the granted buffer is called buffer mode(BM). Each time when a request leaves the granted buffer due to unlocking or transaction rollback, the buffer mode is recomputed. Requests in requesting queue(RQ) may enter the granted buffer if they are compatible with the buffer mode. Each time a request enters the granted buffer, the buffer mode is recomputed by looking up Table 5.

A simple technique for providing starving-free guarantee is first-come-first-service (FCFS). All lock requests for a given object are granted in first come first serve order. FCFS is a simple, fair policy. For example, in Fig. 7, IS*, RS, S, IS modes in the granted buffer are mutually compatible and are granted concurrently. The buffer mode is S*. The request with IX mode in the front of the requesting queue is waiting, because IX is incompatible with the buffer mode S*. Fig. 8 shows the diagram after IS*, S, IS leave the granted buffer. Requests waiting in RQ will try to enter the granted buffer and a new granted buffer is built as depicted in Fig. 9. Locking requests enter the granted buffer according to the sequence of the requests issued.

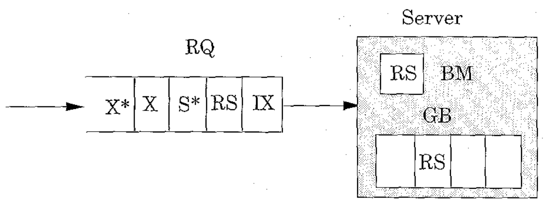


Fig. 8. After IS, S, IS* leave granted buffer.

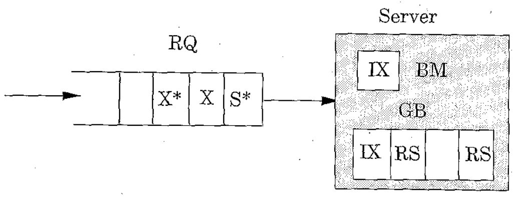


Fig. 9. Building a new scheduling state.

4.5 Dual Queue Scheduling

In some circumstances, FCFS is not good enough when considering the degree of concurrency. For the example, in Fig. 10, shared locks and exclusive locks are interleaved and only one lock request is granted at any moment. There is no concurrency at all in this case.

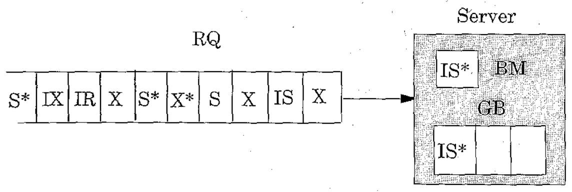


Fig. 10. Degradation caused by FCFS scheduling.

Observing the degradation that can occur with FCFS scheduling, we realize that in certain situations the priority of requests may be changed to increase concurrency. Yet the new policy must be starvation-free. Based on FCFS, the dual queue, scheduling scheme is developed.

The dual queue scheme is similar to two-level feedback

TABLE 5
THE COMBINING MODES FOR MGL

	S	X	S*	X*	IS	IX	IS*	IX*	SIX	SIX*	IR	IW	IRI	IWI	RS	WS
S	S	X	S*	X*	S	SIX	S*	SIX*	SIX	SIX*	S	X	S	SIX	S	WS
X	X	X	X*	X*	X	X	X*	X*	X	X*	X	X	X	X	X	WS
S*	S*	X*	S*	X*	S*	SIX*	S*	SIX*	SIX*	SIX*	S*	X*	S*	SIX*	S*	WS
X*	X*	X*	X*	X*	X*	X*	X*	X*	X*	X*	X*	X*	X*	X*	X*	WS
IS	S	X	S*	X*	IS	IX	IS*	IX*	SIX	SIX*	S	X	IS	IX	IS	WS
IX	SIX	X	SIX*	X*	IX	IX	IX*	IX*	SIX	SIX*	SIX	X	IX	IX	IX	WS
IS*	S*	X*	S*	X*	IS*	IX*	IS*	IX*	SIX*	SIX*	S*	X*	IS*	IX*	IS*	WS
IX*	SIX*	X*	SIX*	X*	IX*	IX*	IX*	IX*	SIX*	SIX*	SIX*	X*	IX*	IX*	IX*	WS
SIX	SIX	X	SIX*	X*	SIX	SIX	SIX*	SIX*	SIX	SIX*	SIX	X	SIX	SIX	SIX	WS
SIX*	SIX*	X*	SIX*	X*	SIX*	SIX*	SIX*	SIX*	SIX*	SIX*	SIX*	X*	SIX*	SIX*	SIX*	WS
IR	S	X	S*	X*	S	SIX	S*	SIX*	SIX	SIX*	IR	IW	IR	IW	IR	WS
IW	X	X	X*	X*	X	X	X*	X*	X	X*	IW	IW	IW	IW	IW	WS
IRI	S	X	S*	X*	IS	IX	IS*	IX*	SIX	SIX*	IR	IW	IRI	IWI	IRI	WS
IWI	SIX	X	SIX*	X*	IX	IX	IX*	IX*	SIX	SIX*	IW	IW	IWI	IWI	IWI	WS
RS	S	X	S*	X*	IS	IX	IS*	IX*	SIX	SIX*	IR	IW	IRI	IWI	RS	WS
WS	WS	WS	WS	WS	WS	WS	WS	WS	WS	WS	WS	WS	WS	WS	WS	WS

queue scheme. The queue in the first level is the requesting queue (RQ) and the queue in the second level is the delaying queue(DQ) as shown in Fig. 11. All lock requests for some class object are appended to the end of RQ. If the request in the front of RQ is incompatible with the buffer mode, the request then is appended to DQ. At the beginning, the system scheduler switches to RQ and the server will serve the requests in RQ. The system scheduler will switch to DQ periodically or when RQ is empty. Each time when the system scheduler switches to DQ, it will not switch back to RQ until the requests in DQ are all served. This guarantees that no requests will be delayed indefinitely. Figs. 11 and 12 show that the scheduler is going to empty DQ. Comparing Fig. 11 and Fig. 10, we can see that the dual queue scheme enhances concurrency. Of course, the size of the queue and the length of the switching period need to be analyzed and tuned for optimal performance.

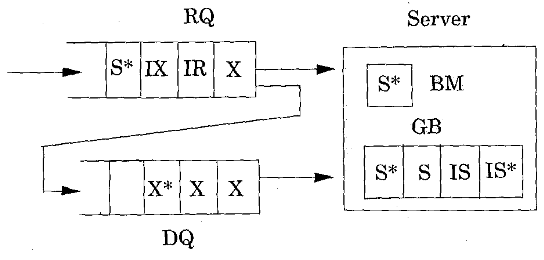


Fig. 11. Requests appended to the delaying queue.

4.6 Comparison of MGL with ORION Locking Model

In general, the efficiency of locking models can be compared in two aspects: degree of concurrency and locking overhead. Comparisons are made between the proposed model(MGL) and the ORION model [13], [12], [15]. Some characteristic query examples are used to illustrate the comparison. These typical examples include queries on schemas, queries on class lattices and queries on composite objects.

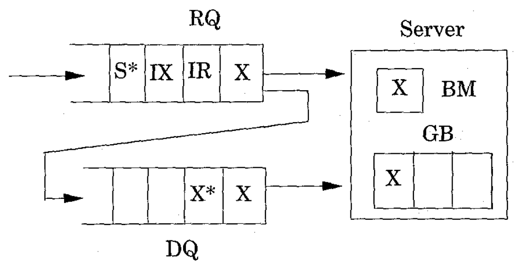


Fig. 12. Emptying the delaying queue.

EXAMPLE 5.1. Add an attribute to class C in Fig. 13.

- *ORION*
  1) Lock all superclasses of class C in shared mode. So classes R, A, and B are locked in S mode.
  2) Lock class C in X mode.
  3) Lock every subclass of class C in exclusive mode. Thus classes D, E, F, and G are locked in X mode.
- *MGL* (Protocol A)
  1) Lock the superclasses of class C along one super-

class chain in RS mode. So, classes A and R (or B and R) are locked in RS mode.
2) Lock class C in WS mode.
3) Lock each subclass of class C with more than one superclass in WS mode. So lock class E in WS mode.

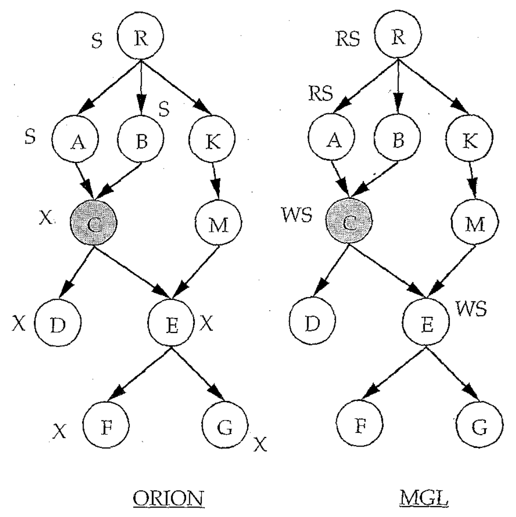


Fig. 13. Comparison: Write schema.

ORION introduces large overhead in locking subclasses and superclasses. Besides, S and X modes lock instances as well as schemas at the same time. The WS mode of MGL locks the schemas only. The WS mode is well understood and unambiguous. The MGL model propagates the intention locks IW along one superclass chain. The locking overhead of ORION is larger than MGL in this example.

EXAMPLE 5.2. Read all the attributes of class C in Fig. 14.

- *ORION*
  1) All superclasses of class C are read. So, classes A, B, and R are locked in S mode.
  2) Lock class C in S mode.
- *MGL* (Protocol A)
  1) Lock the superclasses of class C along one superclass chain in RS mode. So, classes A and R (or B and R) are locked in RS mode.
  2) Lock class C in RS mode.

ORION introduces large locking overhead in locking superclasses. Due to lack of supporting lock modes for reading schema, ORION uses S mode to achieve that. The S mode not only locks the schema of class C but also locks all instances of class C in shared mode implicitly. If there is another query to update some instances of class C, an IX lock on class C will be issued which is conflicting with S. MGL supports RS mode for schema reading. From Table 4, RS mode is compatible with all instance lock modes. Thus MGL has higher degree of concurrency than ORION on reading schemas.

EXAMPLE 5.3. Update all instances of class LandVehicle (single class) that are of red color (Fig. 6).

- *ORION*
  1) Lock class LandVehicle in IX mode.
  2) Lock the selected instances in X mode.

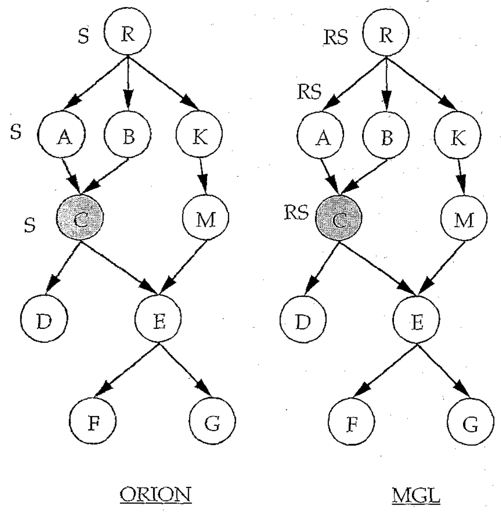


Fig. 14. Comparison: Read schema.

- *MGL* (Protocol B)
  1) Lock the superclasses of LandVehicle along one superclass chain in IWI mode. So class Vehicle is locked in IWI mode.
  2) Lock class LandVehicle in IX mode.
  3) Lock the selected instances in X mode.

ORION is more efficient in locking for single class queries. MGL introduces some overhead in locking superclasses for single class queries. How large the overhead is depends on the number of superclasses.

EXAMPLE 5.4. Update all the instances of class LandVehicle and its subclasses that are of red color (Fig. 6).

- *ORION*
  1) Lock class LandVehicle in IX mode.
  2) Lock each subclass of class LandVehicle in IX mode. That is, lock class RoadVehicle and RailVehicle in IX mode.
  3) Lock the selected instances in X mode.
- *MGL* (Protocol B)
  1) Lock the superclasses of LandVehicle along one superclass chain in IW mode.
  2) Lock class LandVehicle in IX* mode.
  3) Lock the selected instances in X mode.

The locking overhead of ORION mainly depends on the number of subclasses. Each subclass is to be set to IX mode. The locking overhead of MGL is mainly caused by intention locks on superclasses. The overhead of MGL is lower than ORION.

EXAMPLE 5.5. Update the composite object rooted at

Instance[j′] in Fig. 15. Instance[i′], Instance[j′], Instance[k′], Instance[l′], Instance[m′], and Instance[n′] belong to classes I′, J′, K′, L′, M′, and N′, respectively. Instance[j′] and Instance[l′] have shared references on Instance[n′].

- *ORION*
  1) Lock class J′ in IX mode.
  2) Lock composite object Instance[j′] in X mode.
  3) Lock the component class M′ in IXO mode.
  4) Lock the component class N′ in IXOS mode.
- *MGL* (Protocol C)
  1) Lock class J′ in IX mode.
  2) Lock one of Instance[j′]'s parent instance in IX mode. That is, set IX mode to Instance[i′].
  3) Lock composite object Instance[j′] in X mode.
  4) Lock component objects which are shared referenced with X mode. So, lock Instance[n′] in X mode.
  5) Lock the component class M′ in IX* mode.
  6) Lock the component class N′ in IX* mode.

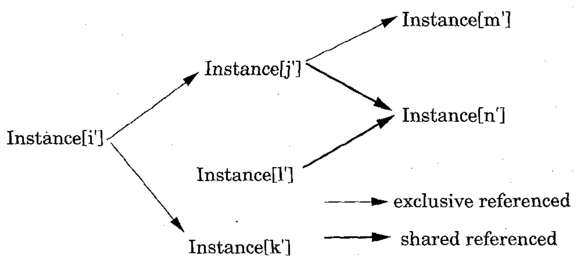


Fig. 15. Example composite objects with shared references.

The protocol of ORION locks the component class N′ in IXOS mode and from Table 1, IXOS mode is incompatible with IS, IX, S, SIX, X modes. The protocol of MGL, however locks N′ also in IX mode and from Table 1, IX mode is only incompatible with S, SIX, X modes. The degree of concurrency of ORION is lower than MGL again.

In general, the locking overhead is somewhat query dependent. From Example 5.3, for single class query, the overhead of ORION is lower, while for schema change in Example 5.1, its overhead is larger. However the degree of concurrency of ORION is never greater than MGL.

5 Summary

The total number of lock modes in the proposed MGL model is sixteen, two modes for schema locking and fourteen modes for instance locking. The lock modes of the MGL models are summarized in Table 6. The properties of the multi-granularity locking model (MGL) are summarized in the following:

- Schema locking is provided.
- Composite object locking is provided.
- More concurrency is gained by adopting a dual queue scheduling scheme.
- A rich set of locking modes is provided. MGL can directly support twelve different operations which are tabulated in Table 7.

TABLE 6
Summary for Locking Modes of MGL

Date	Granule		Lock Modes Allowed	Intention Lock
Schema	Class Lattice		WS	RS
	Single Class		RS	RS
Instance	Class Object	Lattice	S*, X*, IS*, IX*, SIX*	IR, IW, IRI, IWI
		Single	S, X, IS, IX, SIX	IR, IW, IRI, IWI
	Instance Object	Simple	S, X	
		Composite	S, X, IS, IX, SIX	IS, IX

TABLE 7
Operations Directly Supported by MGL

Operations	Lock Mode Supported
read schema of class C	RS
write schema of class C	WS
read all instances of a single class	S
write all instances of a single class	X
read some instances of a single class	IS
write some instances of a single class	IX
read all and write some instances of a single class	SIX
read all instances of a class lattice	S*
write all instances of a class lattice	X*
read some instances of a class lattice	IS*
write some instances of a class lattice	IX*
read all and write some instances of a class lattice	SIX*

By supporting a full set of lock modes for class lattices, MGL has low overhead on locking class lattices. It also gains higher degree of concurrency by applying the hierarchy locking on composite objects and a dual queue scheduling scheme. These results support that the proposed MGL is quite an efficient locking model with higher degree of concurrency. However, the MGL model still can be improved and extended in some ways. Some future works are listed as follows:

- *Performance evaluation*: Although from the comparison examples, we see the proposed MGL model has better degree of concurrency than ORION. A formal and systemic analysis is needed. A performance evalution experiment of the MGL model using simulation technique will be our most important future work.
- *Version*: Versions are often made available in an OODB for supporting many applications, especially in design environments. Objects can be related by the version-derivation relationship. Locking on versionable objects is still an open research issue.
- *More semantic concurrency control*: The semantics employed in the MGL is the relationships between objects only. The semantics of transactions may be used to enhance concurrency further.
- *Nonserializability criteria*: Serializability is a too restrictive criterion for transactions in many applications. It prevents a transaction from seeing the intermediate results of another transaction. For the design environments, designers may need to share uncommitted re-

sults with their co-workers. The additional semantics in object-oriented databases may be used to express flexible correctness criteria other than serializability.

We conclude this paper with a remark that, by utilizing the semantics in object-oriented databases, more efficient locking can be achieved.

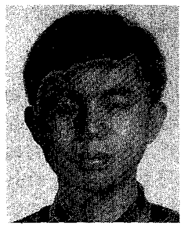
Acknowledgments

This work was supported by the National Science Council, Taiwan, Republic of China, under contract NSC81-0408-E009-02. Special gratitude is extended to the reviewers for their valuable comments and suggestions.

References

[1] J. Banerjee, H.T. Chou, J.F. Garza, W. Kim, D. Woelk, and H.J. Kim, "Data model issues for object-oriented application," *ACM Trans. Office Information Systems*, vol. 5, no. 1, pp. 3-26, 1987.
[2] C. Born, "A formal approach to object-oriented databases," *Comm. ACM*, vol. 10, no. 9, pp. 575-577, Sept. 1973.
[3] P.A. Bernstein, V. Hazilacos, and N. Goodman, "Concurrency control and recovery in database systems. Addison Wesley, 1987.
[4] S.L. Chen, W.P. Yang, and S.Y. Lee, "Granularity of locks in object-oriented database systems," Nat'l Chiao Tung Univ., masters thesis, June 1988.
[5] R.R. Daniel and M. Stonebraker, "Effects of locking granularity in a database management system," *ACM Trans. Database Systems*, vol. 2, no. 3, pp. 233-24, Sept. 1977.
[6] R.R. Daniel, and M. Stonebraker, "Locking granularity revised," *ACM Trans. Database Systems*, vol. 4, no. 2, pp. 210-227, June 1979.
[7] C.J. Date, *An introduction to database systems*. Reading, Mass.: Addison-Wesley, 1990.
[8] J.N. Gray, R.A. Lorie, G.R. Putzolu, and L.I. Traiger, "Granularity of locks and degrees of consistency in a shared database," *Modeling in DataBase Management Systems*, G.M. Nijssen, ed., Elsevier North-Holland, pp. 365-395, 1976.
[9] J.N. Gray, "Notes on database operating systems: Operating systems—An advanced course," *Lecture Notes in Computer Science*. New York: Springer-Verlag, pp. 393-491, 1978.
[10] *Database Engineering* special issue on object-oriented systems, IEEE Computer Soc., vol. 8, no. 4, Dec. 1985,.
[11] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth, "Semantics and implementation of schema evolution in object-oriented databases," *Proc. ACM SIGMOD Int'l Conf. Management Data*, 1987.
[12] J.F. Garza and W. Kim, "Transaction management in an object-oriented database system," *Proc. ACM SIGMOD Int'l Conf. Management Data*, June 1988.
[13] W. Kim, J. Banerjee, H.T. Chou, J.F. Garza, and D. Woelk, "Composite object support in an object-oriented database system," *Proc. Second Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, Orlando, Fla., Oct. 1987.
[14] W. Kim, "Object-oriented databases: Definition and research directions," *IEEE Trans. Knowledge and Data Engineering*, vol. 2, no. 3, pp. 327-340, Sept. 1990.
[15] W. Kim, E. Bertino and J.F. Garza, "Composite objects revisited," *Object-Oriented Programming, Systems, Languages, and Applications*, pp. 327-340, 1990.
[16] H.F. Korth, "Deadlock freedom using edge locks," *ACM Trans. Database Systems*, vol. 7, no. 4, pp. 632-652, Dec. 1982.
[17] H.F. Korth, "Locking primitives in a database system," *J. ACM* vol. 20, no. 1, pp. 55-79, Jan. 1983.
[18] G.T. Nguyen and D. Rieu, "Schema evolution in object-oriented database systems," *Data and Knowledge Engineering*, vol. 4, pp. 43-67, 1989.
[19] A.H. Skarra and S.B. Zdonik, "Concurrency control and object-oriented databases," *Object-Oriented Concepts, Databases, and Applications*, W. Kim, and F. Lochovsky, eds. Reading, Mass.: Addison-Wesley, 1989.
[20] *SQL/Data System: Concepts and Facilities*, GH24-5013-0, File No. S370-50, IBM Corp., Jan. 1981.
[21] R.L. Liou and S.Y. Lee, "A multi-granularity locking model for concurrency control in object-oriented database system," masters thesis, Nat'l Chiao-Tung Univ., Taiwan, 1991.
[22] M. Atkinson, F. Bancilhon, and D. Dewitt, "The object-oriented database system manifesto," *Proc. First DOOD*, pp. 40-57, 1989.
[23] W. Kim and F.H. Lochovsky, *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, ACM Press, 1989.
[24] F. Velez, G. Bernard, and V. Darnis, "The O2 object manager: An overview," *Proc. 15th Conf. Very Large Data Bases*, Amsterdam, The Netherlands, Aug. 1989.
[25] B. Bretl, D. Majer, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams, "The Gemstone data management system," *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.
[26] D. Maier, "Development of an object-oriented DBMS," *Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, Applications*, Portland, Ore., Oct. 1986.
[27] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan, "IRIS: An object-oriented database management system," *ACM Trans. Office Information Systems*, vol. 5, no. 1, pp. 48-69, Jan, 1987.

Suh-Yin Lee received her BSEE degree from the National Chiao Tung University, Taiwan, in 1972, and her MS degree in computer science from the University of Washington, Seattle, in 1975. She joined the faculty of the Department of Computer Engineering at Chiao Tung University in 1976 and received the PhD degree in electronic engineering there in 1982. Dr. Lee is now a professor in the Department of Computer Science and Information Engineering at Chiao Tung University. She chaired the department from 1991 to 1993. Her current research interests include multimedia information systems, object-oriented databases, image/spatial databases, and computer networks. Dr. Lee is a member of Phi Tau Phi, the ACM, and the IEEE Computer Society.



Ruey-Long Liou received his BS degree in electrical engineering from National Central University and his MS degree in computer science from the National Chiao Tung University, Taiwan, in 1989 and 1991, respectively. He has been a systems engineer at the Institute for Information Industry, Taiwan, since 1991. His major work focuses on the areas of object-oriented databases, concurrent control, and object-oriented programming. He is also interested in Global Positioning System applications and embedded systems.