An Effective Class Hierarchy Concurrency Control Technique in Object-Oriented Database Systems

Woochun Jun and Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73019
wocjun@cs.ou.edu; gruenwal@cs.ou.edu

Abstract

In this paper, we present a locking-based concurrency control scheme for object-oriented databases (OODBs). Our scheme deals with class hierarchy which is an important property in OODBs. The existing concurrency controls for a class hierarchy perform well only for specific environments. Our scheme is based on so called special classes and can be used for any applications with less locking overhead than existing works, for both single inheritance and multiple inheritance. In this study, we show the superiority of our scheme over existing works.

keywords: concurrency control, class hierarchy, object-oriented database

1. Introduction

While conventional databases are suitable for modeling simple data type, object-oriented databases (OODBs) are good for managing non-standard applications such as computer-aided design (CAD) and computer-aided software engineering (CASE). These new applications require advanced modeling power in order to handle complex data and complex relationships among data. This can be extremely difficult or even impossible in conventional business-oriented applications. In an OODB, a class object consists of a group of instance objects and class definition objects. The class definition consists of a set of *attributes* and *methods* that read or modify attributes of an instance or a set of instances. Users can access objects by invoking methods. A typical user transaction consists of a set of method invocations on objects [3].

In general, there are two types of access to an object: *instance access* methods (instance read method and instance write method) and *class definition access* methods (class definition read method and class definition write method) [1,3]. An instance access consists of consultations and/or modifications of attribute values in an instance or a set of instances. A class definition access consists of consultations of class definition and/or modifications of class definition such as adding/deleting an attribute or a method.

Commutativity is a criterion widely used to determine whether a method can run concurrently with methods in progress on the same object. Two methods commute if their execution orders are not important (i.e., their execution orders do not affect their results). Two methods conflict with each other if they do not commute.

Concurrency control is a mechanism used to coordinate access to the multi-user database so that the consistency of the database is maintained [2,12]. A concurrency control scheme allows multi-users rapid access to a database but incurs an overhead whenever it is invoked. This overhead may have a critical effect on OODBs where many transactions are long-lived. Thus, reducing the overhead is vital to improve transaction response time.

One of the major properties of an OODB is inheritance. That is, a subclass inherits definitions defined on its superclasses. Also, there is an *is-a* relationship between a subclass and its superclasses. Thus, an instance of a subclass is a specialization of its superclasses (and conversely, an instance of a superclass is a generalization of its subclasses) [10]. This inheritance relationship between classes forms a *class hierarchy*. While there are some operations on only one class such as class definition read or on only one instance such as instance write, there are two types of operations on a class hierarchy: *instance access to all or some instances of a given class and its subclasses* which we call IACH (Instance Access to Class Hierarchy) and *class definition write*. A query is an example of IACH where a *query* is defined as an instance read to a given class and its subclasses [6]. Due to inheritance, the definitions of a class' superclasses should not be modified, while the class and its instances are being accessed. Also, due to the *is-a* relationship between classes, the search space for a query against a class, say C, may include the instances of all classes in the class hierarchy rooted at C as well as instances of C. Thus, for a locking based concurrency control scheme, when a class definition write or query is requested on some class, say C, it is necessary to get locks for all subclasses of C as well as C. We call MCA (Multiple Class

Access) for class definition write and IACHs, and SCA (Single Class Access) for an operation to only one class such as class definition read and instance access to a single class.

In this paper, we present a locking-based concurrency control scheme for class hierarchy in OODBs. There are two approaches in the literature that deal with class hierarchy, *explicit locking* and *implicit locking*, both will be discussed in Section 2. These approaches may work well only for specific applications in OODBs. Explicit locking may have less locking overhead for transactions concerned only with SCA. On the other hand, implicit locking may have less locking overhead for transactions concerned only with MCA. Our scheme is based on a so called *special class*, which will be defined in Section 3, and which can be used for any applications with less locking overhead. With an assumption that the number of access is stable for each class, we show that our scheme performs better than both explicit locking and implicit locking.

This paper is organized as follows. In Section 2 we review previous studies. In Section 3 we present our class hierarchy locking scheme. In Section 4 we show how to assign special classes for a given class hierarchy and prove the superiority of our work to existing works. The paper concludes with plans for future research in Section 5.

2. Related Work

As discussed in Section 1, due to inheritance, dass definition writes and IACHs on a class may need to access more than one class on a class hierarchy. There are two major existing approaches to perform locking on a class hierarchy: *explicit locking* [3,11] and *implicit locking* [6,9,10]. In *explicit locking*, for an IACH involving a class, say C, and all of its subclasses, and for a class definition write on a class C, a lock is set not only on the class C, but also on each subclass of C on the class hierarchy. For other types of access such as class definition read and instance access to a single class, a lock is set for only the class to be accessed (we call *target class*). Thus, for an MCA, transactions accessing a class near the leaf level of a class hierarchy will require fewer locks than transactions accessing a class near the root of a class hierarchy. Another advantage of *explicit locking* is that it can treat *single inheritance* where a class can inherit the class definition from one superclass, and *multiple inheritance* where a class can inherit the class definition from more than one superclass, in the same way. However, this technique increases the number of locks required by transactions accessing a class at a higher level in the class hierarchy.

In *implicit locking*, setting a lock on a class C requires extra locking on a path from C to its root as well as on C. *Intention* locks [4,8] are put on all the ancestors of a class before the target class is locked. An intention lock on a class indicates that some lock is held on a subclass of the class. For MCA on a target class, locks are not required for every subclass of a target class. It is sufficient to put a lock only on the target class[1] (in single inheritance) or locks on the target class and subclasses of the target class which have more than one superclass (in multiple inheritance) [10]. Thus, it can reduce lock overhead over explicit locking. But, *implicit locking* requires a higher locking cost when a target class is near the leaf level in the class hierarchy due to intention lock overhead.

For example, consider the following class hierarchy. Note that we select two OODBs, Orion [6] and O_2 [3], for the illustration of two existing class hierarchy techniques. In order to update the class definition in class, say C, each scheme works as in Fig. 1.a. Here, for *implicit locking*, intention locks IWs corresponding to W (Write) locks are required for all superclasses on the path from C to the root A. Thus, if another transaction, say T1, needs to update the class definition in A (i.e., it needs to get W lock on class A), it does not have to search through each class in the class hierarchy because the help of the intention lock IW on class A. That is, since IW and W conflict each other, T1's lock request is blocked on class A. In implicit locking, there is no conflict between intention locks, and between an intention lock and an SCA lock; however, there can be a conflict between an intention lock and an MCA lock depending on the commutativity relationship. On the other hand, an explicit locking does not require any intention locks. But, it does require a Cw (Class Write) lock on each subclass (i.e., class D and E) of the target class through the class hierarchy since any modification of class definitions in C may affect the definitions of its subclasses. Also, locking for a query on C (assuming that all instances of C, D and E) can be done as in Fig. 1.b.

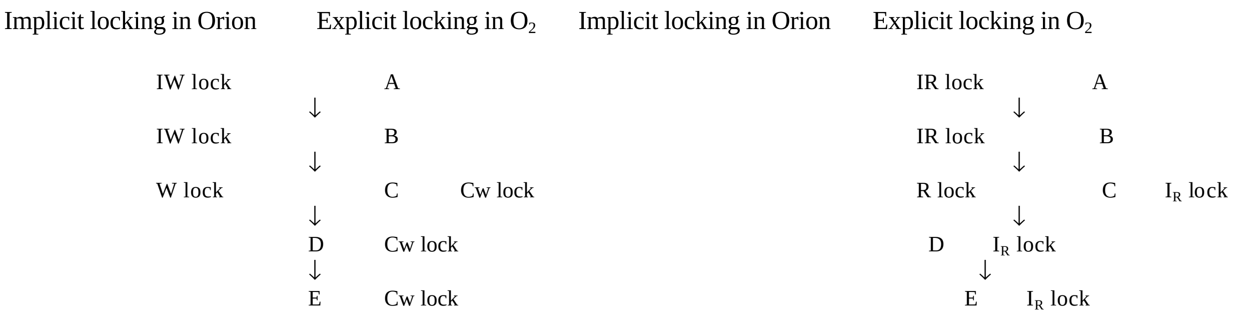


Fig. 1.a. locking for class definition write in Orion and O_2 Fig. 1.b. locking for query in Orion and O_2

3. Proposed class hierarchy locking scheme

3.1. Basic idea

Our work is to develop a new class hierarchy locking scheme which can be used for any OODB applications with less locking overhead than both existing schemes, explicit locking and implicit locking. To achieve this, we designate some classes in the class hierarchy as *special classes*. Roughly, we define a *special class* (SC) as a class on which class definition writes or IACHs are performed frequently. For our concurrency control purposes, how to determine if a class is a SC or not will be discussed in Section 4.

In this new class hierarchy locking scheme, intention locks are set on SCs only; thus, locking overhead is reduced compared to implicit locking which requires intention locks on every superclass of the target class. When a transaction needs to access an SC which is already intention locked, by invoking an MCA lock on it, a concurrency control can reduce conflict checking overhead due to the help of the

[1] Note that, for a query to some instances of a class and its subclasses, locks are required for those individual instances in each subclass.

intention lock. That is, every conflict can be detected by the help of commutativity relationships between intention locks and MCA locks on the SC. On the other hand, if a class has little or no possibility of being accessed by an MCA, there is no need to set an intention lock on that class since SCAs do not use intention locks to check for conflicts. As we discussed earlier, there is no conflict between an intention lock and an SCA lock and any conflict is determined only at target class. Thus, unlike implicit locking, we do not have to set an intention lock on every class on the path from a target class to a root.

In order to have fewer locks required for an MCA than those required by explicit locking, our scheme works as follows: for an SCA, a lock is set on only the target class, like explicit locking. For MCAs, unlike explicit locking which requires locks on the target class and all its subclasses, locks are set on every class from the target class to the first SC through the subclass chain of the target class. If there is no such SC, then locks are set on leaf classes. If the target class is an SC, then set a lock only on the target class. Thus, as we have discussed, by choosing an SC as a class on which MCAs are performed frequently, we can reduce locking overhead.

For example, consider the following class hierarchy in Fig. 2.a. Assume a transaction T1 invokes an MCA lock on class C6. Let LS1 be a lock setting for T1. Assuming that classes C1, C4 and C7 are SCs, then Fig. 2.b, 2.c, and 2.d show how locks are set in explicit locking, implicit locking, and our scheme.

$$\begin{tikzcd}
\text{C1} \arrow[d] & \text{C1} \arrow[d] & \text{C1: LS1} \arrow[d] & \text{C1(SC): LS1} \arrow[d] \\
\text{C2} \arrow[d] & \text{C2} \arrow[d] & \text{C2: LS1} \arrow[d] & \text{C2} \arrow[d] \\
\text{C3} \arrow[d] & \text{C3} \arrow[d] & \text{C3: LS1} \arrow[d] & \text{C3} \arrow[d] \\
\text{C4} \arrow[d] & \text{C4} \arrow[d] & \text{C4: LS1} \arrow[d] & \text{C4 (SC): LS1} \arrow[d] \\
\text{C5} \arrow[d] & \text{C5} \arrow[d] & \text{C5: LS1} \arrow[d] & \text{C5} \arrow[d] \\
\text{C6} \arrow[d] & \text{C6: LS1} \arrow[d] & \text{C6: LS1} \arrow[d] & \text{C6: LS1} \arrow[d] \\
\text{C7} \arrow[d] & \text{C7: LS1} \arrow[d] & \text{C7} \arrow[d] & \text{C7 (SC): LS1} \arrow[d] \\
\text{C8} \arrow[d] & \text{C8: LS1} \arrow[d] & \text{C8} \arrow[d] & \text{C8} \arrow[d] \\
\text{C9} \arrow[d] & \text{C9: LS1} \arrow[d] & \text{C9} \arrow[d] & \text{C9} \arrow[d] \\
\text{C10} & \text{C10: LS1} & \text{C10} & \text{C10}
\end{tikzcd}$$

Fig. 2.a	Fig 2.b	Fig 2. c	Fig. 2.d
Class hierarchy	Explicit locking	Implicit locking	Our scheme

3.2. Lock Modes

We adopt instance level granularity for instance access and entire class object for class definition access, like Orion [6] and O_2 [3]. Below we show locks needed for different types of instance and class access [7]. For convenience, we use lower-case letters and upper-case letters to name locks on an instance and a class, respectively.

Operations	Locks needed
instance read	r (on target instance) TR, IMPR, INTSR, QR (on target class or its superclasses)
instance write	w (on target instance) TW, IMPW, INTSW, QW (on target class or its superclasses)
Class definition write	CW (on target class) INTSW (intention lock for each SC on the path from the target class to its root)
Class definition read	CR (on target class) INTSR (intention lock for each SC on the path from the target class to its root)

• instance read

- (for SCA) TR (Target Read) lock means that some (not all) instances of a target class are *r* locked. A TR lock is set on a *target class* whenever an *r* lock is set on its instance.

- (for SCA) IMPR lock (Implicit Read on target class) means that all instances are *read* locked *implicitly.* Like both explicit locking and implicit locking, we reduce locking overhead by setting an IMPR lock on the target class, not on individual instances, if the majority of instances are accessed.

- (for MCA) QR (Query Read on a target class) means that some or all instances of a target class and its subclasses are *read* locked as in implicit locking. We reduce locking overhead by setting an QR lock on only the target class, not setting TR or IMPR lock on the all subclasses of the target class.

- An intention lock INTSR (Intention Superclass Read) is set for every SC on the *superclass chain* from a target class to its root whenever an IMPR or TR or a QR lock is set on the target class. It indicates that some instance read lock is held on a subclass of the class.

• Instance write

- (for SCA) TW lock (Instance Write on target class) means that some (not all) instances of a target class are *w* locked. An TW lock is set on a *target class* whenever *w* lock is set on its instance.

- (for SCA) IMPW (Implicit Write) lock means that all instances of a target class are *w* locked *implicitly*.

- (for MCA) QW (Query Write on a target class) means that some or all instances of a target class and its subclasses are write locked.

- INTSW (Intention Superclass Write)lock is set for every SC on the *superclass chain* from the target class to its root whenever an IMPW or TW or QW lock is set on an instance or class.

3.3. Commutativity Relation Tables

In Tables 1 and 2, we provide commutativity relation among the lock modes introduced above. Y(Yes) and N (No) stand for *commute*, and *not commute*, respectively.

a) instance

		lock holder	
		r	*w*
lock	*r*	Y	N
requester	*w*	N	N

Table 1. Commutativity relation for locks on an instance

b) Class

As in~~ implicit locking, conflicts between MCAs, if at least one of the lock ho~~lder and requester requires locks only on a class, conflict relationships are determined directly by read-read, read-write, write-write conflict. Otherwise (i.e., both require locks on a class as well as instances), conflicts are determined on individual instances. For conflicts between MCAs and intention locks, conflicts are determined as if an intention lock were a real lock. For example, setting CW and INTSR on the same class will cause conflicts. Also, there is no conflict between SCAs and intention locks.

		lock holder									
		CW	CR	TR	IMPR	INTSR	QR	TW	IMPW	INTSW	QW
r	CW	N	N	N	N	N	N	N	N	N	N
e	CR	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
q	TR	N	Y	Y	Y	Y	Y	Y	N	Y	N
u	IMPR	N	Y	Y	Y	Y	Y	N	N	Y	N
e	INTSR	N	Y	Y	Y	Y	Y	Y	Y	Y	N
t	QR	N	Y	Y	Y	Y	Y	N	N	N	N
e	TW	N	Y	Y	N	Y	N	Y	N	Y	N
r	IMPW	N	Y	N	N	Y	N	N	N	Y	N
	INTSW	N	Y	Y	Y	Y	N	Y	Y	Y	N
	QW	N	Y	N	N	N	N	N	N	N	N

Table 2. Commutativity table for locks on a class

3.4. Class hierarchy locking algorithm for single inheritance

Our locking-based concurrency control scheme is based on two-phase locking which requires each transaction to obtain a read (or write) lock on a data item before it reads (or writes) that data item, and not to obtain any more locks after it has released some lock [5]. For a given lock request on a class, say C, we set locks on C and all classes on the class hierarchy to which the class C belongs as follows. For simplicity, we adopt strict two-phase locking [12] which requires each transaction to release all the locks at the commitment time.

Step 1) locking on SCs

• For each *SC* (if any) through the superclass chain of C, check conflicts and set an *intention lock* if it commutes. If it does not commute, block the lock requester.

Step 2) Locking on a target class

•If the lock request is an SCA, check conflicts with locks set by other transactions and set one of TR, TW, IMPR, IMPW (depending on the lock request type) or CR (class definition read) on only the target class C if it commutes and set an *r* or *w* lock on the instance to be accessed (which we call *target instance*) if *a* method is invoked on the instance and commute. If it does not commute, block the requester.

• If the lock request is an MCA, then, from class C to the first *SC* (or leaf class if there is no SC) through the subclass chain of C, check conflicts and set either CW, QR, or QW lock on each class if commute. If the class C is an SC, then set a lock only on C.

Note that we set a lock on the first SC so that we let other incoming transactions that access a subclass of the first SC check conflicts since those transactions need to set intention locks on the first SC. Thus, every conflict can be detected. The reason we set a lock on each class (besides the first SC) from the class C to the first SC (not including the SC) is as follows: if a lock is set only on the first SC, then some conflict may not be detected. For example, if a requester accesses a subclass of a lock holder's class which is CW locked, then such a conflict may not be detected.

•If class C has more than one subclass, perform the same step 2) for each subclass chain of C.

As an example, consider the following lock requests by two transactions T_1 and T_2 on a class hierarchy in Fig. 3.a

a) T_1: class definition write (CW) request on class C6

b) T_2: class definition read on class C5

Let LS_i be a lock LS set by transaction T_i. Assume that class C1, C4 and C7 are SCs. As seen in Fig 3.b, 3.c, and 3.d, 7, 12 and 11 locks are required for T_1 and T_2 by our scheme, explicit locking, and implicit locking, respectively.

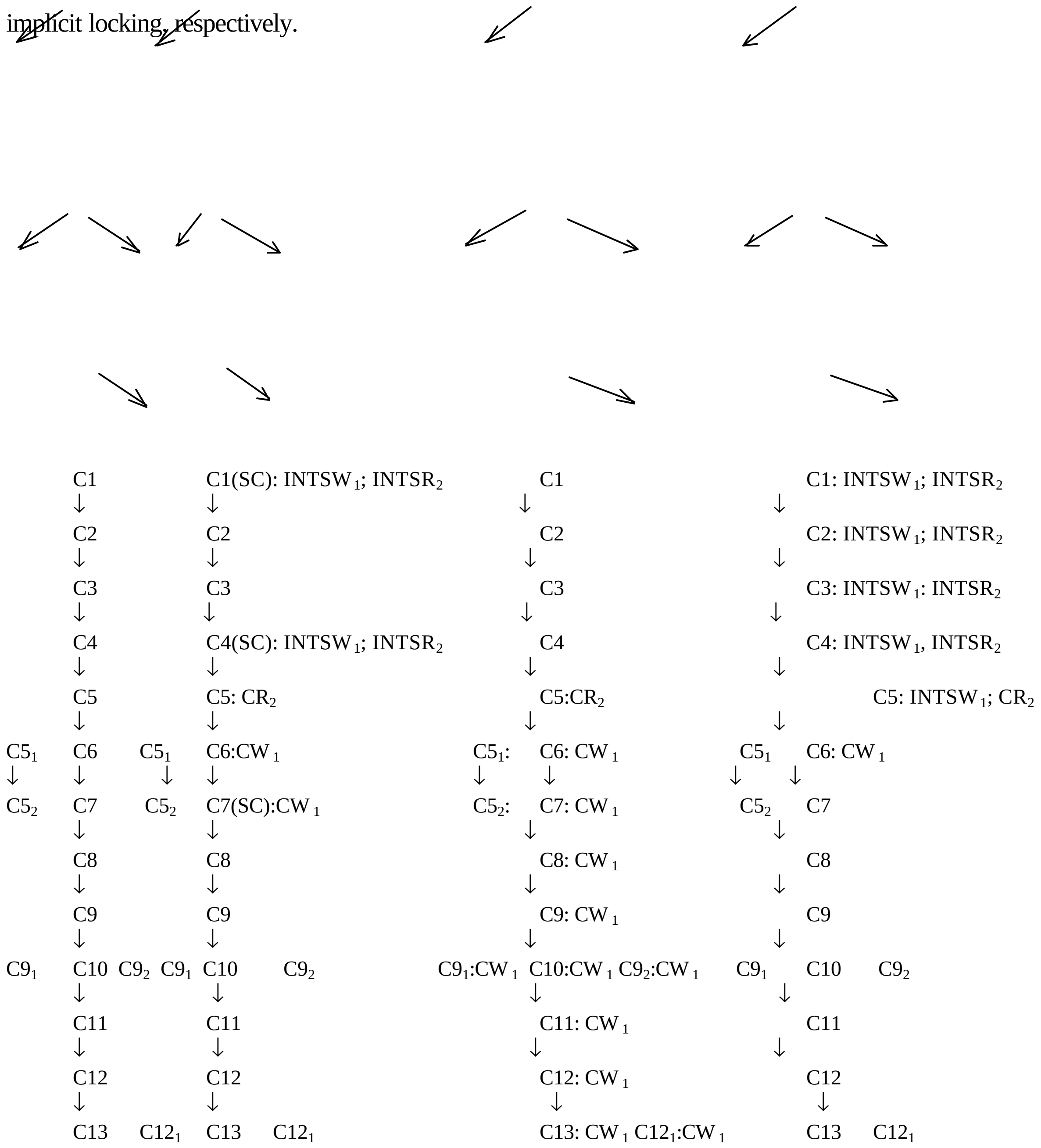


Fig 3.a class hierarchy Fig. 3.b. Locks with Our scheme Fig. 3.c. Locks with Explicit locking Fig. 3.d. Locks with Implicit locking

3.5. Considering Multiple Inheritance

The above protocol works for a *single inheritance* class hierarchy. But, it does not work for a *multiple inheritance* class hierarchy. Consider the following class hierarchy in which a class J has two superclasses, H and I in Fig. 4. a. Suppose a transaction T1 sets a QR lock on class F (Fig. 4. b). Suppose now another transaction T2 sets a CW lock on class G (Fig. 4.c). Even though T1 already sets a conflict lock mode QR on class J and K implicitly, T2 can get a lock successfully since intention locks on class A which is only common class requiring locks by both transactions do not conflict. That is, the above protocol does not work correctly.

In order to make the above protocol work correctly in *multiple inheritance*, we adopt the principle from Orion [6]: when setting a QR, PQR, QW, PQW, or CW lock on a class C, also set those locks on subclasses of C which have more than one superclass. Then only those subclasses need to be examined for conflict checking. Also, we set intention locks on each class through only one superclass chain of the target class. In this example, using our scheme, lock settings for QR lock on class F are changed as in Fig. 4.d.

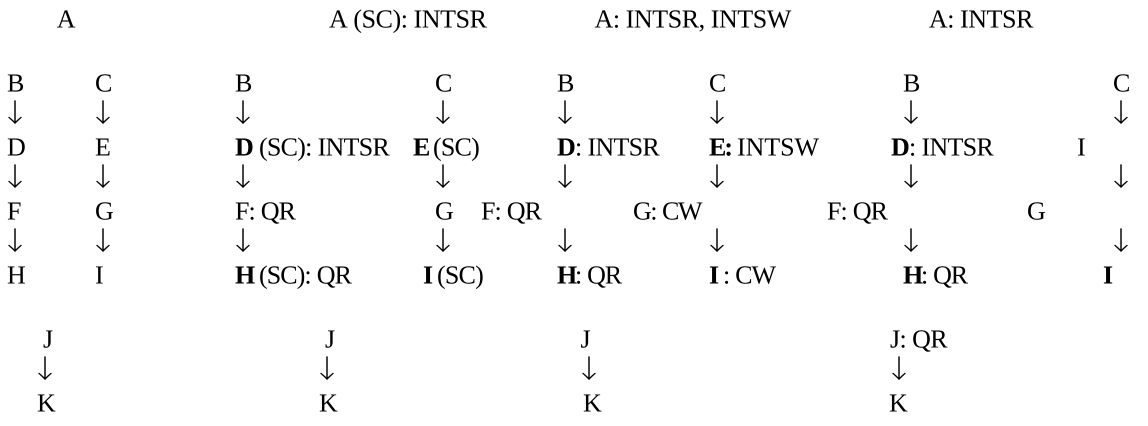


Fig. 4.a class hierarchy

Fig. 4.b QR lock on class F

Fig. 4.c CW lock on class G

Fig. 4.d Locks with multiple inheritance consideration in our scheme

3.6. Correctness of our scheme

Now, we prove that our algorithm is correct, that is, it satisfies serializability [5]. We prove this by showing that, for any lock requester, its conflict with a lock holder (if any) is always detected. With this proof, since our class hierarchy locking scheme is based on two-phase locking, it is guaranteed that our scheme satisfies serializability [5]. Also, for simplicity, we prove only for single inheritance. For multiple inheritance, the correctness can be proved similarly as in single inheritance. If a lock requester is an SCA, then its lock holders (whose lock modes need to be checked for conflict with lock requester) consist of transactions holding locks on the target class and all special classes in the superclass chain of the target class. If a lock requester is an MCA, then its lock holders include those defined above plus transactions holding locks on each class from the target class to the first special class in the subclass chain of the target class.

There are four cases depending on the types of lock requesters and holders.

case 1*) the lock holder is an SCA*
the lock requester is an SCA

If a lock holder (L_H) and a lock requester (L_Q) access different classes, there is no conflict. If a lock holder and a lock requester access the same class, there is no conflict on all special classes through the superclass chain of the target class because intention locks on SCs are compatible with L_Q ; conflicts can be detected on the target class (for example, IMPW and TR) or target instance (for example, TR and TW).

case 2) the lock holder is an SCA
the lock requester is an MCA

If the L_H is holding a lock on a superclass of the L_Q's class, there is no conflict since the L_Q does not access the L_H's class. If the L_H is holding a lock on the L_Q's class or subclass, then there are two subcases. If there exists an SC between L_H and L_Q, then conflict is detected on the nearest SC through the subclass chain of the L_Q's class (case 2.1). Otherwise, the conflict is detected on the class of L_H (case 2.2). Let R and H be two classes on which the L_Q requests a lock and the L_H holds a lock, respectively. In case 2.1, as shown in Fig. 5.a, a conflict (if any) is checked on SC1, which is the nearest *special class* of the L_Q's class through its subclass chain, since the L_H has an intention lock on SC1 and the requester requests CW, QR or QW on SC1. On the other hand, in case 2.2, for subcase a, a conflict (if any) is checked on H as in Fig. 5.b since the L_H does not have any intention locks through the superclass chain of R and the L_Q needs to set a CW, QR or QW lock on H. For subcase b, conflict (if any) is checked on H as in Fig 5.c since intention locks on all special classes through the superclass chain of H are compatible and the requester needs to set a CW, QR or QW lock on H.

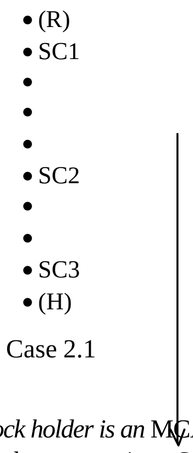


Fig 5. a. Case 2.1

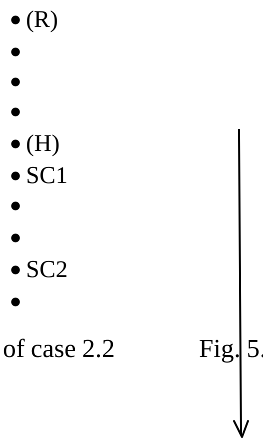


Fig. 5.b. Subcase a of case 2.2

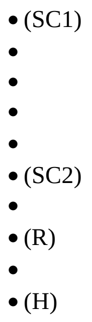


Fig. 5.c. Subcase b of case 2.2

case 3) the lock holder is an MCA
the lock requester is an SCA

If the L_H is holding a lock on a subclass of the L_Q, there is no conflict. If L_H is holding a lock on the class of L_Q or on a superclass of L_Q, then there are two cases in which conflicts will be detected. If there exists some SCs between L_Q and L_H, the conflict is detected on the nearest SC to L_H through the subclass chain of L_H such as SC2 in Fig. 6.a (case 3.1). Otherwise, conflict is detected on the class of L_Q as in Fig. 6.b (case 3.2).

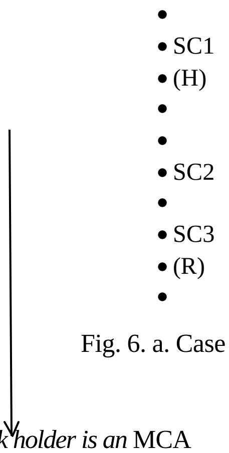


Fig. 6. a. Case 3.1

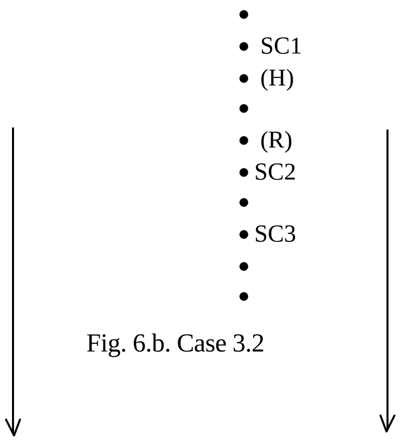


Fig. 6.b. Case 3.2

case 4) the lock holder is an MCA
the lock requester is an MCA

If the L_H accesses the same class or superclass of the L_Q's class, the conflict is detected as in either case 3.1 or case 3.2. On the other hand, if the L_H accesses subclass of the L_Q' class, the conflict is detected as in either case 2.1 or case 2.2.

From cases 1), 2), 3) and 4), we can conclude that, for any lock requester, it is guaranteed that its conflict with a lock holder (if any) is always detected. Also, since our scheme is based on two-phase locking, serializability is guaranteed [5].

4. Special Class Assignment and Its Performance Evaluation

4.1. Special class assignment

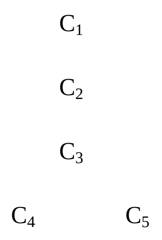
Assume that we have information on the number of access to each class (by different transactions) in an OODB. For our scheme, we need to know only two types of number of access to each class: SCA and MCA. With this number of access information for each class, we determine if the class is designated as an SC or not as follows.

```
Starting from each leaf class until all classes are checked.
step 1) If a class is a leaf, then do not designate it as an SC.
     If a class C has not been considered for SC assignment and all subclasses of C have been already
     considered (i.e., they have been determined for SC assignment), then do the followings
          for class C and all of the subclasses,
               calculate the number of locks ($N_1$) when the class is designated as an SC
               calculate the number of locks ($N_2$) when the class is not designated as an SC
               // In calculation, we do not consider any superclass of C yet. This is due to the bottom-up
               // approach in our SC assignment. That is, for each class C, we perform SC assignment
               // only for C, and all subclasses of C.

step 2) Designate it as an SC only if $N_1 < N_2$ . That is, the class can be an SC only if the number of locks
can be reduced by doing so.
```

For example, consider a simple class hierarchy as in Fig 7.a and assume that we have number of access information on the hierarchy as in Fig. 7.b. The numbers represent the numbers of access to the class by different transactions. For example, in Fig. 7.b, 100 MCAs are performed on class C_1 and 300

SCAs on C_1, by different transactions accessing this class hierarchy. Note that, for MCAs, the numbers represent only access initiated at a given class. Thus, we do not count MCA access numbers initiated at its superclasses. In our SC assignment scheme, since C_4 and C_5 are leaf classes, they are not designated as SCs. At the class C_3, if C_3 is designated as an SC, the number of locks needed for class C_3, C_4 and C_5 are 450 (for C_3), 300 (for C_4) and 400 (for C_5), respectively, resulting 1150 locks for the three classes. That is, C_3 needs only 450 locks since any locks are not necessary for its subclass as in implicit locking. On the other hand, any access C_4 and C_5 needs intention locks on C_3, resulting 300 and 400 locks for C_4 and C_5, respectively. On the other hand, if C_3 is not designated as an SC, then the total number of locks needed for classes C_3, C_4, and C_5 are 1200 locks, where 850 locks are for C_3 (800 locks for MCA and 50 locks for SCA), 150 locks are for C_4 (100 locks for MCA and 50 locks for SCA), and 200 locks are for C_5 (100 locks for MCA and 100 locks for SCA). In this case, our scheme works as in explicit locking. Thus, class C_3 becomes an SC. Similarly, two classes C_1 and C_2 become non-SCs. Fig. 7.c shows the result of our SC assignment scheme based on access frequency information.



C_1 : MCA:100, SCA: 300

C_2 : MCA: 200, SCA: 200

C_3 : MCA:400, SCA: 50

C_4 : MCA: 100, SCA: 50

C_5 : MCA:100, SCA: 100

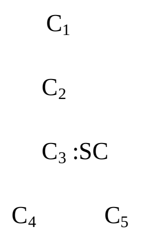
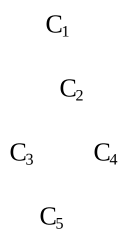


Fig. 7.a. Simple class hierarchy Fig. 7.b. Access numbers for each class Fig. 7.c. Result of SC assignment

For multiple inheritance, the same SC assignment scheme can be applied. For example, consider a simple multiple class hierarchy as in Fig. 8.a. and assume that we have frequency information on the hierarchy in Fig. 8.b. Assume that, when C_5 is locked, C_3 is chosen for intention lock setting. The result of SC assignment scheme is shown in Fig. 8.c.



C_1: MCA: 50, SCA: 100

C_2 : MCA: 600, SCA: 200

C_3 : MCA: 100, SCA:150

C_4 : MCA: 300, SCA: 100

C_5 : MCA: 100, SCA: 50

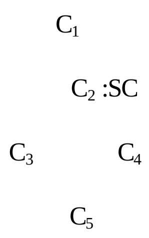


Fig. 8.a. Simple class hierarchy Fig 8.b. Access numbers for each class Fig. 8.c. Result of SC assignment

4.2. Performance evaluation of our scheme

In this subsection, we show that our scheme performs better than both explicit locking and implicit locking. That is, assuming that the number of access is stable for each class, we show that our scheme incurs either equal or fewer number of locks than both explicit locking and implicit locking. Our proof is based on induction.

Claim: with stable number of access for each class, our class hierarchy scheme performs better than both explicit locking and implicit locking.

Proof) We use induction on the number n in a given class hierarchy. Let n be the number of classes considered in SC assignment scheme so far. Let N_E, N_I and N_O be the number of locks by explicit locking, implicit locking and our locking, respectively, for classes considered in SC assignment so far.

- n =1 : $N_E = N_I = N_O$
- n=2 : In this case, without loss of generality, two classes are formed as follows.

$$\begin{tikzcd} C_1\text{ (SC)} \arrow[d] \\ C_2\text{ (leaf)} \end{tikzcd}$$
case a)

$$\begin{tikzcd} C_1 \arrow[d] \\ C_2\text{ (leaf)} \end{tikzcd}$$
case b)

If C_1 (superclass) is an SC as in case a), then $N_O \leq N_E$ otherwise C_1 would not be an SC, and $N_O = N_I$. If C_1 is not an SC as in case b), similarly $N_O \leq N_I$, and $N_O = N_E$.

Assume that our scheme works up to n = K

• n = K+1: without loss of generality, we can consider (K+1)th class as a root of the classes considered for SC assignment. Let x be a root (i.e., (K+1)th class) and $y_1 \ldots y_m$ be the first m SCs through the subclass chains of x as in Fig. 9. Also, Let N (x:SC) and N (x:non-SC) be the numbers of locks required in our scheme when a class x is designated as an SC and x is not designated SC, respectively (we assume that all subclasses of x have been considered in the SC assignment scheme).

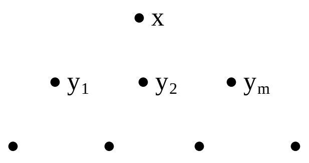


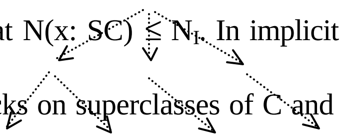
Fig. 9. The case where x is not SC

case a) Assume that x is not SC (i.e., N (x:SC) > N (x:non-SC))

We first prove that $N_O \leq N_E$: for locks required for SCA to class x, both schemes need the same number of locks. For MCA to class x, for our scheme, locks are required for each class from x to $y_1 \ldots y_m$ and subclasses of x which have more than one superclass, if multiple inheritance. On the other hand, locks are required from x to every subclass of x by explicit locking. Also, for locks required for access to classes other than x, $N_O \leq N_E$ by induction assumption and no intention locks on x are necessary by our scheme. Thus, $N_O \leq N_E$ for any access.

Now, we prove that $N_O \leq N_I$.

subcase a.1), $N_O \leq$ N(x:SC). Otherwise, x would be an SC by our SC assignment scheme.



subcase a.2) We prove that N(x: SC) $\leq N_I$. In implicit locking, every class is an SC since any access to class C needs intention locks on superclasses of C and any MCA access to C need no locks other than a lock on class C. This is corresponding to our scheme where all classes are SCs. Thus, for locks required for access to x, both schemes incur the same number of locks. For locks required for access to other than x, intention locks (if necessary) are needed to be set on x by both schemes. But, for locks required for access to classes other than x, $N_O \leq N_I$ by induction assumption. Thus, N (x :SC) $\leq N_I$. This implies that $N_O \leq N_I$.

case b) Assume that x is SC (i.e., N(x:SC) < N(x:non-SC))

$N_O \leq N_I$: same as subcase a.2

Now, we prove that $N_O \leq N_E$.

subcase b.1), $N_O \leq$ N(x:non-SC). Otherwise, x would not be a SC by our SC assignment scheme.

subcase b.2) N(x:non-SC) $\leq N_E$: for locks required for SCA to class x, both schemes incur the same number of locks. For MCA to class x, locks are needed from x to $y_1 \ldots y_m$ as in Fig. 9 and subclasses of x which have more than one superclass, if multiple inheritance, in our scheme. But locks are required from x to every subclass of x in explicit locking. For locks required for access to classes other than x, $N_O \leq N_E$ by induction assumption. Thus, $N_O \leq N_E$.

From cases a) and b), with stable number of access to a class hierarchy, we can conclude that our scheme incurs less or at least equal number of locks than both explicit and implicit locking.

5. Further work

In this paper, we present a concurrency control scheme for class hierarchy in OODBs. Our scheme is based on *special class* in order to reduce locking overhead. With assumption that number of access for each class is stable, our scheme always incurs fewer number of locks than both explicit locking and implicit locking, for both single inheritance and multiple inheritance.

In order to compare our work with both implicit locking and explicit locking in real environment, we will conduct simulation using some representative OODB benchmark. Also, in our work, locking granularity is instance object and class object for instance access and class definition access, respectively.

We are currently developing a class hierarchy locking scheme with finer granularity. That is, we are adopting attribute level granularity instead of instance, and finer class definition instead of entire class object, in order to provide better concurrency among transactions.

References

[1] Agrawal D. and Abbadi A, A Non-restrictive Concurrency Control for Object-Oriented Databases, 3rd Int. Conf. on Extending Data Base Technology, Vienna, Austria, (Mar. 1992), 469 - 482.

[2] Bernstein P, Hadzilacos V and Goodman N, *Concurrency Control and Recovery in Database Systems*, (Addison-Wesley, 1987).

[3] Cart M and Ferrie J, Integrating Concurrency Control into an Object-Oriented Database System, 2nd Int. Conf. on Extending Data Base Technology, Venice, Italy, (Mar. 1990), 363 - 377.

[4] Date, C., *An Introduction to Database Systems*, Vol. II, (Addison-Wesley, 1985).

[5] Eswaran K, Gray J, Lorie R and Traiger I, The notion of consistency and predicate locks in a database system, Communication of ACM, Vol. 19, No. 11, (Nov., 1976), 624 - 633.

[6] Garza J and Kim W, Transaction Management in an Object-Oriented Database System, ACM SIGMOD Int. Conf. on Management of Data, Chicago, Illinois, (Jun., 1988), 37 - 45.

[7] Jun W and Gruenwald L, A Flexible Class Hierarchy Locking Technique in Object-Oriented Database System, 11th Int. Conf. on Computers and Their Applications, San Francisco, (Mar., 1996) 191 - 196.

[8] Korth H and Silberschartz A, *Database System Concepts*, 2nd Edition, (McGraw Hill, 1991).

[9] Lee L and Liou R, A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems, IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 1, (Feb. 1996), 144 - 156.

[10] Malta C and Martinez J, Controlling Concurrent Accesses in an Object-Oriented Environment, 2nd Int. Symp. on Database Systems for Advanced Applications, Tokyo, Japan, (Apr., 1992) 192 - 200.

[11] Malta C and Martinez J, Automating Fine Concurrency Control in Object-Oriented Databases, 9th IEEE Conf. on Data Engineering, Vienna, Austria, (Apr., 1993), 253- 260.

[12] Ozsu M and Valduriez P, Principles of Distributed Database Systems, (Prentice Hall, 1991).