Semantic-Based Concurrency Control in Object-Oriented Databases

Woochun Jun and Le Gruenwald
Dept. of Computer Science
Univ. of Oklahoma
Norman, OK 73019

Abstract

In this paper, we present a concurrency control mechanism that deals with three important issues in object-oriented databases (OODBs): *semantics of methods*, *nested method invocation* and *referentially shared object*. In our scheme, locks are required for the execution of methods instead of atomic operations. By doing this, we reduce the locking overhead and deadlocks due to lock escalation. Especially, if a method invokes one or more methods on the same object during its execution, our scheme does not incur additional overhead for those invoked methods. Also, we provide a way of automating commutativity of methods. In addition, we further increase concurrency with the use of run-time information.

1. Introduction

Object-oriented databases (OODBs) have been adopted for non-standard applications. Examples of such applications include computer-aided software engineering (CASE), computer-aided design (CAD), and office information systems which require advanced modeling power to handle complex data and relationships among such data.

OODBs are a collection of classes and instances of these classes. In OODBs, both classes and instances are referred to as *objects*. A class object consists of a set of attributes and methods through which the class's instances are accessed. Users can access objects by invoking methods. In order to make sure of the atomicity of user interactions, the traditional transaction model can be used in an OODB. That is, users can access an OODB by executing transactions, each of which is defined as a partially ordered set of method invocations on a class or an instance object [Agra92]. 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 do not affect the results. Two methods conflict with each other if they do not commute.

One of the important characteristics of database systems is manipulation of shared data. That is, database systems, including OODBs, allow shared data to be accessed by multiple users at the same time. A concurrency control involves synchronization of access to the database so that the consistency of the database is maintained [Ozsu91, Bern87]. Serializability is a widely used criterion of correctness. Transactions are *serializable* if the interleaved execution of their operations produces the same results as some serial execution of the same transactions [Bern81].

Supporting concurrency control in an OODB is more complicated than in a relational database for the following reasons. First, the semantics of methods on encapsulated objects can be exploited to provide better concurrency. That is, when we consider the semantics of two methods, those two methods may run concurrently even though they conflict in terms of read and write access modes. But this semantic analysis may not be done in an automatic way for many applications since those semantics are drawn only by application programmers' discretion. Second, in an OODB, a method invoked on a higher level object can invoke another method defined on a component object of the higher level object. This nested method invocation fits naturally with the nested transaction model [Moss85]. This model provides many useful features such as increased parallelism, localized failure and reusable partial results. As a result, nested transaction models are being used for transaction management in OODBs [Agra92, Muth93, Rese94]. Finally, referentially shared objects (RSOs) are a fundamental concern in OODBs since new objects may be composed from existing objects. The referentially shared objects may share common subobjects in an underlying hierarchy [Herr90]. Thus, method invocations on different objects may not commute due to possible conflicts on subobjects. [Rese94] states "such conflict may not be determined a priori and defining conflict relationships based on static analysis becomes very conservative". From this statement,

we conclude that static analysis results in severe degradation in concurrency . Thus, for RSOs, conflict relations should be determined for dynamic method execution.

In this work, we present a locking-based concurrency control protocol for OODBs which addresses all the issues: *semantics of methods*, *nested method invocations*, and *RSOs*. We reduce overhead due to frequent invocation of concurrency control protocol as well as deadlocks due to lock escalation. Our locking scheme requires locks for methods only so that we do not need commutativity tables for atomic operations invoked during a method execution. We also reduce locking overhead due to one or more methods invoked during a method execution on the same object. In addition, we provide a way of automating commutativity of methods. Usually, locks by method invocations provide less concurrency than locks by atomic operations [Malt93]. We overcome this obstacle by the use of run-time information.

This paper is organized as follows. In the Section 2, we present previous studies and discuss their advantages and disadvantages. In Section 3, we discuss our locking scheme. The paper concludes with further work in Section 4.

2. Related Work

In earlier attempt for nested method invocations, a locking technique is developed for disjoint and non-disjoint complex object in [Herr90]. They argue that the traditional approaches dealing with complex object have the following problems : *the granule-oriented problem, protocol oriented problem* and *authorization problem.* Locking entire complex object may decrease concurrency severely although it can reduce concurrency control overhead. On the other hand, lock individual object can lead to tremendous overhead (granule-oriented problem). In non-disjoint complex objects, updating shared object

can lead big overhead since all parent objects of the shared object should be locked (protocol-oriented protocol). Combining concurrency control and authorization component can achieve higher concurrency (authorization-oriented problem). For example, if a transaction does not have right to update some object, an exclusive lock is not required for the object. In order to solve three problems above, for a complex object type, they created the *general lock graph*, which is to solve the granule-oriented problem. In turn, for the general lock graph, the *corresponding object-specific lock graph* which is to solve protocol-oriented problem and authorization-oriented problem, is constructed. Although their locking protocol considers non-disjoint subobject (RSO), they do not exploit any semantics in order to provide higher concurrency.

In existing OODBs such as Orion, O_2 and Gemstone [Garz88, Cart90, Serv90], they do not exploit the nested behavior of OODB transaction in order to increase concurrency. This is due to that their locking is based on standard flat transaction model. But, nested method invocations naturally fit a nested transaction model as follows: for a nested transaction, there is a top-level transaction which consists of a sequence of steps, where each step is either a primitive operation or the invocation of a subtransaction. Each subtransaction may also consist of either or both primitive operations and subtransactions. Thus, in OODB, a method invocation corresponds to a subtransaction in nested transaction model. For this reason, the nested transaction model is adopted for nested method invocations.

. The notions of nested transactions for database systems and a nested two-phase locking protocol are introduced in [Moss85]. In [Moss85], locks are only required for the execution of atomic operations encountered during a method execution such as reading an attribute. A method execution cannot terminate until all of its children are terminated. When a method execution terminates, its parent inherits its locks. Locks are discarded only if the top-level transaction terminates. Their work has simplicity in implementation, but it does not take advantage of any semantics of methods, which results in limited concurrency.

The nested two-phase locking with *ordered sharing* is proposed in [Agra92]. Their work is based on nested two-phase locking in [Moss85] and locks are required for each atomic operation. They provide better concurrency using *ordered sharing* between locks. Unlike commutativity relationships, when ordered sharing is adopted, a lock request is never delayed until a transaction holding a conflicting lock commits or aborts. That is, a lock request is always granted as follows: for a given operation o_1, the set of operations are divided into two categories: the set of operations that commute with o_1, and the set of operations that do not commute with o_1. If o_1, a lock requester, commutes with some operation o_2, a lock holder, (i.e., o_1 has a *shared relationship* with respect to o_2), the lock request is granted and the execution order between them is not important. But if o_1 does not commute with o_2 (i.e., o_1 has *an ordered shared relationship* with o_2), a lock request is granted but the execution order should be preserved by observing so called *ordered commit rule*: if a transaction T_i (which consists of a set of *operations* and the *relation* ordering conflicting operations in T_i) is granted a lock with an ordered shared relationship with respect to a lock held by T_j on an object and T_j is a proper descendent of parent of T_i., then, T_i cannot commit unless T_j has committed or aborted. Also, like a nested transaction model in [Moss85], a transaction cannot commit or abort until all its children are terminated, and locks are inherited by its parent when it commits. Even though [Agra92] increases concurrency among methods by adopting *ordered sharing*, they do not exploit semantics of methods.

In [Muth93], a locking-based concurrency control scheme for OODB is presented. They exploit the semantics of methods to increase concurrency. In their work, the conflict between lower level operations or methods can be ignored due to the commutativity of higher level invoked methods in nested method execution. In their work, a lock is required on an object whenever a method or operation is called on the object. Also, locks are converted to *retained lock* at the end of subtransaction. If a top-level transaction is commits, all the locks held are released. They use semantics of methods as follows: when two atomic operations conflict with each other, if they have a commutative ancestor pair and the ancestor

of the lock holder commits, the lock request is granted. That is, the lock request is not delayed until top-level transaction commits so that high degree of concurrency can be achieved. Similarly, when two methods conflict with each other, the same principle can be applied. But, these authors do not consider OODBs with RSOs. This is a weakness of their work because RSOs are a fundamental property of OODBs and are necessary for modular design as indicated in [Rese94].

A semantic two-phase locking protocol for OODB is presented in [Rese94]. They consider RSOs and nested method execution. Also, they use semantics of method in order to increase concurrency. In this work, locks are required only for atomic operations. The protocol works as follows: a subtransaction or top-level transaction *T* cannot terminate until all of the children are terminated. When a subtransaction is committed, its locks are inherited by its parent. On the other hand, when a transaction is aborted or is top-level and committed, its locks are released. A lock request is granted if one of the three following conditions are met: (a) no other transaction holds a conflicting lock, (b) if there are conflicting locks, such locks are held by its ancestors and (c) if there are conflicting locks held by non-ancestors of lock holders, then one of the ancestors of the lock holders (not including the lock holders) and some ancestors of the lock requester commute. Locking for each atomic operation incurs an overhead which has a critical effect on OODBs where many transactions are long-lived. This results in more deadlocks due to lock escalation which is a main source of deadlock [Malt93]. Note that a lock is said to be *escalated* if more exclusive lock is requested for an object by the same transaction during its execution. Also, [Rese94] assumes that the commutativity relationships between methods are well-defined and can be derived based on semantics as well as the specification of the class and its methods. But, [Rese94] fails to provide a formal way to construct commutativity relationships among methods.

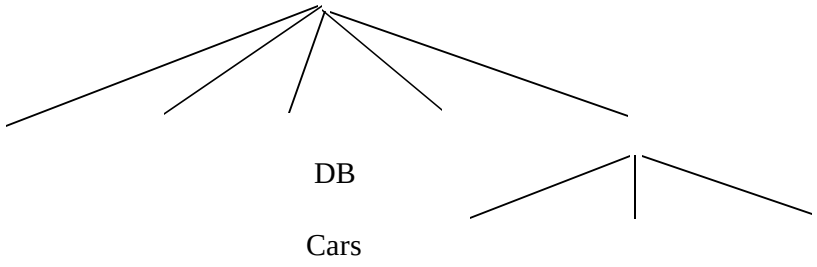
3. Our Scheme

In this section, we present a locking scheme dealing with all three aspects discussed earlier: *semantics of methods, nested method invocation and referentially shared objects.*

3.1. Assumption

We assume that objects are organized in a hierarchy and referential sharing is allowed. Also, we adopt the following transaction model and method model: a transaction consists of a sequence of method invocations to objects [Cart90, Agra92]. A method execution consists of a partial order of method invocations and atomic operations [Hadz91]. Also, we assume that a method in an object can invoke methods on objects which are lower in the hierarchy [Rese94].

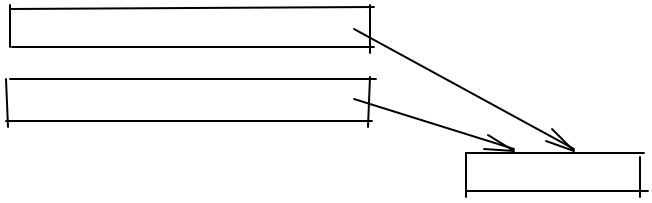
Consider the following object hierarchy in Fig. 1.a. The database (DB) consists of class *Cars*. Each *car* instance is a tuple object composed of various atomic objects and of component class *Orders*. Each *order* instance is a tuple object composed of atomic objects. In our scheme, referential sharing is allowed. That is, an instance of class *Order* can be shared by different instances of class *Cars*. In this object hierarchy, we assume that a customer can rent only one car at any time. But a customer can request multiple car rental orders so that the order is granted by any available car. Fig 1.b shows an example of a car rental order requested for two cars by a customer.



Car-id Name Price-To-Rent QOH Orders
(Quantity-on-hand)

Order-No Customer-No Status

Fig. 1.a. An Object Hierarchy



Car:

i1 Grand-Am $30 4 •

i2 Grand Prix $50 2 •

Order: 10 1 new

Fig. 1.b. An example of the object hierarchy

Assume that three are three methods *Adjust-Price*, *Check-Out-Rent* and *Pay-Rent*, for class *Cars*.

Adjust-Price(i)

// For a car instance i (Car-id), if QOH is greater than 10, price to rent a car is decreased by 10% //

If i.QOH > 10 then

    i.Price-To-Rent <= i.Price-To-Rent * 0.9

End if

End

Check-Out-Rent(i,Order-No)

// For a car instance i, a rent-a-car request by Order-No o is granted if that order is not granted yet //

If Test-status (o) = *new* then

    call Change-Status (o, granted)

    i.QOH <= i.QOH -1

end if

End

Pay-Rent (i,o)

// Pay rental fee for car i by Order-No o //
read i.Price-To-Rent
read i.QOH
Change-Status (o, *paid*)
End

For class Orders, assume that two methods *Test-status* and *Change-status* have the following implementation code, respectively. There are three status for each order: *new*, *granted* and *paid*.

Test-status (o)
// test status of an instance o of class Orders //
read (o.status)
return status
End

Change-status(o, value)
// change status of an instance o of class Orders to *value* //
write (o.status, *value*)
End

3.2. Automation of commutativity for methods

In this subsection we present a way to automate commutativity relationships among methods. Thus, we do not put a burden on application programmers to specify method commutativity. Usually, locks by methods may provide less concurrency than locks by atomic operations [Malt93]. In order to overcome this shortcoming, we increase concurrency by making use of dynamic information.

The automation of commutativity relationships among methods is based on the notion of *affected sets of attributes* [Badr88, Jun95]. That is, even if two methods conflict in terms of read or write operations, as long as their access modes on individual attributes do not conflict, those two methods can

run in parallel. A Direct Access Vector (DAV) [Malt93] is constructed for each method and also for break points (which will be defined later) in the method. A DAV is a vector whose fields correspond to attributes defined in the class on which the method operates. Each value composing this vector denotes the most restrictive access mode used by the method or break point when accessing the corresponding field. An access mode of an attribute can have one of three values, N (Null), R (Read) and W (Write) with N < R < W for their restrictiveness. Two methods commute if their corresponding DAVs commute, that is, their access modes are compatible for each attribute.

For example, assume that we have two methods MT1 and MT2, and a class Y with four attributes a_1, a_2, a_3, and a_4 as follows.

method MT1	**method MT2**
read a_1	read a_1
If (a_1 > 100) then	read a_2
a_3 := a_1	If (a_1 > a_2) then
end if	return a_1
If (a_2 > 10) then	else
a_4 := a_2	return a_2
end if	end if

The DAVs constructed for method MT1 and MT2 are: DAV(MT1) = [R,R,W,W] and DAV(MT2) = [R,R,N,N]. Since access modes are compatible for each attribute, the two methods, MT1 and MT2, commute.

We need a two-phase pre-analysis which consists of two steps : 1) construction of DAV for each method and 2) construction of a commutativity table of methods. In each method, a break point is inserted by a programmer or a compiler when a conditional statement is encountered. Every method has a special break point called *first break point* before the first statement in the method. There are three kinds of DAVs in each method : 1) a DAV of the entire method, 2) a DAV of the first break point, which is a

union of access modes of each attribute used by statements that are executed regardless of execution paths 3) a DAV of every other break point, which contains access modes of all attributes used by statements between this break point and the next break point (or end of the method). A union operation denoted as '⊕' takes two arguments among N (null: no operation), R (Read), and W (Write) and selects the more restrictive one.

Table 1 illustrates how the union operation works. Note that, if one or more method for the same object is defined in a breakpoint, DAV of the breakpoint includes DAV of the method defined.

	N	R	W
N	N	R	W
R	R	R	W
W	W	W	W

Table 1. Union operation table

For example, consider the object hierarchy in Fig. 1.a. For convenience, for class *Cars*, let four attributes Car-id, Name, Price-To-Rent and QOH be a_1, a_2, a_3, and a_4, respectively. Similarly, for class *Orders*, let three attributes Order-No, Customer-No, and Status be b_1, b_2, and b_3, respectively. Assume that, for class Cars, A and A_1 are breakpoints of method *Adjust-Price*, B and B_1 are breakpoints of method *Check-Out-Rent* and C is a breakpoint of method *Pay-Rent.* Likewise, assume that, for class *Orders*, let D and E be breakpoint of methods *Test-Status* and *Change-Status*, respectively. Also, for simplicity, we call methods *Adjust-Price*, *Check-Out-Rent* and *Pay-Rent* as M1, M2 and M3, respectively. Similarly, we call methods *Test-status* and *Change-status* as N1 and N2, respectively, for class *Orders*.

Adjust-Price (also called M1)

[A]

If i.QOH > 10 then

[A1]

i.Price-To-Rent := i.Price-To-Rent*0.9

End if

End

Check-Out-Rent(i,Order-No) (also called M2)

[B]

If Test-status (o) = *new* then

[B_1]

call *Change-Status* (o, *granted*)

i.QOH := i.QOH - 1

end if

End

Pay-Rent (i,o) (also called M3)

[C]

read i.Price-To-Rent

read i.QOH

Change-Status (o, *paid*)

End

Based on our definition of breakpoints and DAVs, for the object hierarchy in Fig. 1.a, we have the following breakpoints and DAVs for the methods. Let DAV(x) represent DAV of a breakpoint x or a method x.

Note that, in this example, a method *Check-Out-Rent* or *Pay-Rent* includes another nested method invocation (*Test-status* or *Change-status*). But, this nested method invokes another object so that its DAV is not included in the DAV of method *Check-Out-Rent* or *Pay-Rent.*

The DAVs constructed for method M1 are:

DAV (M1) = [R,N,W,R]; DAV (A) = [R,N,N,R]; DAV (A1) = [R,N,W,N]

Similarly, the DAVs for M2 and M3 are:

DAV (M2) = [R,N,N,W]; DAV (B) = [R,N,N,N]; DAV (B1) = [R,N,N,W]

DAV (M3) = [R,N,R,R]

Similarly, for class *Orders*, we have DAVs of each break point in the method as follows.

Test-status (o) (also called N1)
[D]
read (o.status)
End

Change-status(o, value) (also called N2)
[E]
write (o.status, value)
End

DAV (N1) = [R,N,R]; DAV (N2) = [R,N,W]

Note that, for class *Orders*, two methods do not have conditional statements so that DAVs of the methods are the same as DAVs of the first breakpoints. In our work, we do not include DAVs of the first break point for such a case.

After the construction of the breakpoints' DAVs in all methods, we create a commutativity relation of methods in the form of a table. In this table, a lock requester's entries contain names of DAVs of all methods (denoted as D(Mi) where Mi is the name of the method). For example, D(M1) represents a

DAV of the method M1, which is [R,N,W,R]. A lock holder's entries contain names of DAV of each method (denoted as D(Mi)), name of the DAV of the first break point in Mi (denoted as D(K) if K is the first breakpoint) and names of the DAVs of other break points (denoted as $D(K_j)$ if K_j is the breakpoint other than first break point in Mi). For example, in method M1, D(M1), D(A), and $D(A_1)$ represent the following DAVs [R,N,W,R], [R,N,N,R] and [R,N,W,N], respectively. Since we assume the worst case access mode for each attribute before execution, lock requesters always have the most restrictive access modes (i.e., DAVs of methods). But, after a method execution, a lock holder may have a less restrictive access mode (i.e., DAV of the first and/or other break points). Two break points commute if their corresponding DAVs commute. Two DAVs commute if, for each attribute, its access modes in the two DAVs commute. Fig. 2 gives the commutativity tables constructed in our scheme. Note that Y means commute, and N not commute.

		lock holders						
		D(M1)	D(A)	D(A1)	D(M2)	D(B)	D(B1)	D(M3)
lock	D(M1)	N	Y	N	N	Y	N	N
requester	D(M2)	N	N	Y	N	Y	N	N
	D(M3)	N	Y	N	N	Y	Y	Y

Figure 2.a. A commutativity table for class *Cars*

		lock holders	
		D(N1)	D(N2)
lock	D(N1)	Y	N
requester	D(N2)	N	N

Fig. 2.b. A commutativity table for class *Orders*

Our concurrency control is based on two-phase locking [Eswa76]. When a transaction invokes a method on an object, a lock is required for the method (i.e., D(Mi) lock is required). As the transaction meets a break point during run-time, the break point is recorded in a transaction manager. After the method execution, the lock is changed from D(Mi) to D(K), $D(K_1)$,...$D(K_n)$ where D(K) is the name of the DAV of the first break point in Mi and $D(K_1)$,,,$D(K_n)$ are the names of the DAVs of other break points encountered during the method execution Mi. Since the union of DAVs of D(K), $D(K_1)$,...$D(K_n)$ may be less restrictive than the DAV of Mi, that is, D(Mi), this can give more concurrency to other transactions which request locks on the same object. For example, assume that a transaction T1 invokes a method M1 on instance i1 of class Cars and has break points A after the execution of M1. Assume that another transaction T2 comes and invokes a method M3 on the same instance i1 while T1 still has a lock on i1. Applying our technique gives commutativity between M1 and M3 since D(M3) commutes with D(A) by the commutativity table in Fig. 2.a. This commutativity would not be possible if we adopted method commutativity relationship without run-time information checking. In our work, when a DAV of a method or breakpoint, say M, contains DAV of other methods, say M_1 ,,, M_n for the same object, we do not have to request locks for methods M_1 ,,, M_n. That is, one lock request by M is enough for the object so that we can reduce the lock overhead.

In our work, a method may have many breakpoints depending on the method's logic. This results in larger commutativity tables and incurs much run-time overhead for lock changes and conflict checking. Ways to reduce the number of breakpoint are presented in [Jun95].

3.3. Considering semantics, nested method invocation and RSO

In this subsection, we present a way of dealing with three aspects of our concurrency control: *semantics of methods*, *nested method invocation* and *RSO* (Referentially Shared Object).

At first, based on automated commutativity relationships presented in Section 3.2, we allow application programmers to define commutativity relationships for some methods by making use of semantics of methods as in [Muth93, Rese94]. Thus, though these two methods do not commute in terms of read and write access modes, they may commute semantically at the discretion of an application programmer. For example, for class *Cars*, two methods *Check-Out-Rent* and *Pay-Rent* may commute semantically, that is, customer may check out first and then pay the rental fee or vice versa. If two methods, say, M1 (requester) and M2 (holder), commute semantically, then we give *S* commutativity relationship between M1 (and all breakpoints of M1) and M2 (and also all breakpoints of M2) where *S* means semantically commute. Then, we can have a new commutativity table for class *Cars* as in Fig. 3. In the commutativity table, Y means commute (unconditionally). That is, if two methods (one is a lock requester and the other is a lock holder) have Y relationship, a lock requester can get a lock at any time. If two methods have N relationship, a lock requester can get a lock only if the lock holding transaction is committed or aborted. On the other hand, if two methods have S relationship, a lock requester can get a lock if a holder's method execution is finished. That is, the requester does not have to wait until the lock holding transaction is committed or aborted.

		lock holders						
		D(M1)	D(A)	$D(A_1)$	D(M2)	D(B)	$D(B_1)$	D(M3)
lock requester	D(M1)	N	Y	N	N	Y	N	N
	D(M2)	N	N	Y	N	Y	N	S
	D(M3)	N	Y	N	S	S	S	Y

Figure 3. A commutativity table for class *Cars*

For nested method invocations, we have the following principles: each method invocation is associated with a lock. Before any method invocation, a lock is requested and granted. Also, when a method execution is finished, the lock is inherited by its parent. Then, the lock is said to be *retained* by its

parent [Rese94]. If a transaction is finished, its locks are discarded. For two methods which commute semantically, two methods commute only if both execute atomically. That is, for such methods, a requester cannot get a lock until a holder's method execution is finished so that the requester can get a lock only if a holder's lock is inherited by its parent. Thus, unlike N commutativity relationship, a lock request is not delayed until the lock holding transaction commits.

Finally, for RSOs, method invocations on different objects may result in conflicts since those methods may invoke methods on the same subobject. In our scheme, conflicts are determined dynamically for each subobject as in [Rese94] since such a conflict may not be detected a priori based on the static analysis. If such a conflict is defined statically, concurrency may be decreased severely.

3.4. Our concurrency control scheme

Our concurrency control scheme is based on two-phase locking [Eswa76]. Based on our discussion in Section 3.3, we have the following locking scheme.

1. Lock is required only for method execution and is granted before method execution. After method execution, lock is changed (i.e., it reflects the breakpoints executed)
2. A method execution cannot terminate until all of its children are terminated. When a method execution *m* terminates:
    a. there exits parent of *m* and *m* commits : locks held by *m* are inherited by its parent (i.e., locks are *retained* by its parent)
    b. either there exits parent of *m* and *m* aborts or there is no parent of m: locks held by *m* are discarded.
3. A lock can be granted if either of the following conditions is satisfied.
    a. no other method holds or retains a conflicting lock

b. if conflicting locks are held, such locks are retained by ancestors of the requesting method

c. (for semantic commutativity) if conflicting locks are retained by non-ancestors, then one of the ancestors of the retainer (not including the retainer itself) and an ancestor of the requester commute

In rule 3.b, when we allow ancestor/descendent parallelism, we do not let a parent see uncommitted results of the child method. For example, assume that a parent T initiates a method M, which accesses some data item X, and continues to do its own work. When T needs to access data item X so that it requires a conflicting lock on X, T can get a lock only if the lock held by M is retained by T.

In rule 3.c, we implement semantically commutativity relationships. As we discussed for the two methods which commute semantically, two methods commute only if both execute in an atomic way. Thus, we let a lock requester get a lock only if a holder's method execution is finished (i.e., its lock is inherited by an ancestor). In additions, for two methods commuting semantically, a requester's descendent can also get a lock if a holder's method execution is finished.

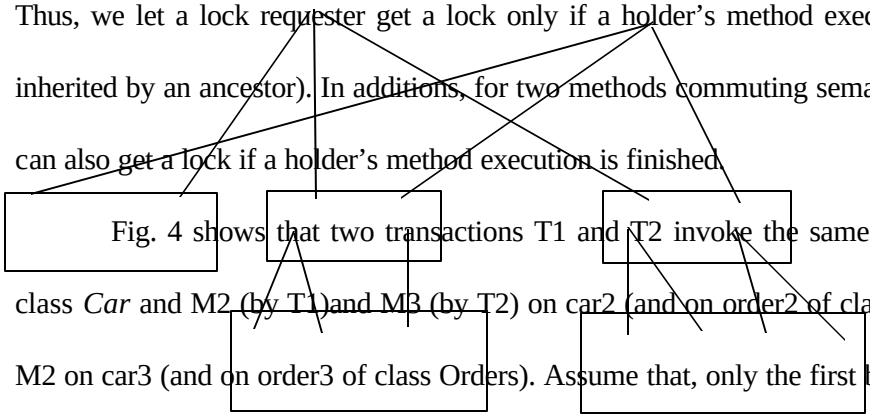


Fig. 4 shows that two transactions T1 and T2 invoke the same method M1 on instance car1 of class *Car* and M2 (by T1)and M3 (by T2) on car2 (and on order2 of class Orders), and the same method M2 on car3 (and on order3 of class Orders). Assume that, only the first breakpoint [A] has been executed in two method invocations of M1 by T1 and T2 on an instance car1 and breakpoints [B] and [B_1] have been executed on an instance car3 in method invocations of M2 by T1 and T2.

time

Fig. 4. A possible execution of transactions in our scheme

In the above example, two method invocations of M1 on car1 commute by adopting dynamic information. This commutativity would not be possible if we used static commutativity relationships for methods as in [Malt93]. Also, two methods M2 and M3 on instance car2 commute semantically so that the method invocation on M3 by T2 can be executed only after M2 invoked by T1 is finished, that is, after the lock held by M2 is inherited by T1. This guarantees atomic execution of method invocation M2 by T1. Without the semantics of methods, the method execution M3 by T2 is blocked until entire transaction T1 is committed. In our scheme, the method execution is delayed only until method M2 invoked by T1 is committed. Thus, we can increase concurrency by adopting semantic information. Also, a lock request by method invocation M2 of T2 on car3 is not granted since a conflicting lock is held by T1. Thus, the method invocation M2 of T2 can be executed after T1 is committed. For method invocations on instance car3, consider the following execution which requires locks by atomic operations as in [Rese94]. This results in deadlock situation as in Fig. 5. In our scheme, we can avoid such a deadlock situation by adopting locks for the execution of the methods.

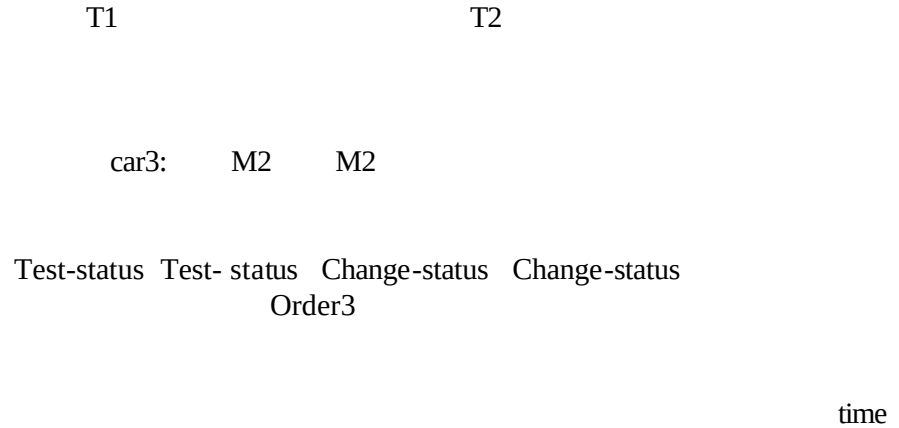


Fig. 5. A possible execution by a scheme requiring locks for atomic operations

For the correctness of our protocol, we can prove it as follows: for every execution resulted from our protocol, we always can find an equivalent serial execution by a series of two transformations: substitution and commutativity-based reversal [Beer89, Rese94]. We start from the bottom level of execution and apply transformations so that we have only serial top-level transactions. The formal proof is similar to the technique in [Rese94] and for that reason we omit here.

4. Further Work

In this paper, we present a locking-based concurrency control scheme in OODBs. Our scheme deals with three important issues in OODB concurrency control: *semantics of methods*, *nested method invocations*, and *referentially shared object.* In our scheme, locks are required for each method invocation so that we reduce the locking overhead and possible deadlock due to lock escalation. In order to overcome the problem of less concurrency due to locks by method invocations, we adopt a scheme to provide run-time information to increase concurrency.

In our scheme, we do not consider inheritance hierarchy in which a subclass inherits or redefines definitions of superclass of the subclass. Currently, we are developing an integrated concurrency control for both object hierarchy and inheritance hierarchy. We will also consider class definition modification in our scheme.

5. References

[Agra92] D. Agrawal and A. E. Abbadi, "A Non-restrictive Concurrency Control for Object-Oriented Databases", 3rd Int. Conf. on Extending Database Technology, Vienna, Austria, Mar. 1992, pp. 469 - 482.

[Bard88] B. Badrinath and K. Ramamritham, “Synchronizing Transactions on Objects”, IEEE Transaction on Computers, Vol. 37, No. 5, pp. 541 - 547, 1988

[Beer89] C. Beeri, P. A. Bernstein, and N. Goodman, “A Model for Concurrency in Nested Transactions Systems”, Journal of the ACM, Vol. 36, No. 2, Apr. 1989, pp. 230 - 269.

[Bern81] P. A. Bernstein and N. Goodman, “Concurrency Control in Distributed Database Systems”, ACM Computing Surveys, Vol. 13, No. 2, Jun. 1981, pp. 185 - 221.

[Bern87] P.A. Bernstein, V. Hadzilacos and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[Cart90] M. Cart and J. Ferrie, “Integrating concurrency control into an object-oriented database system”, 2nd Int. Conf. on Extending Data Base Technology, Venice, Italy, Mar. 1990, pp. 363 - 377.

[Eswa76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, “The notion of consistency and predicate locks in a database system”, Communication of ACM, Vol. 19, No. 11, Nov. 1976, pp. 624 - 633.

[Garz88]. J. F. Garza and W. Kim, “Transaction Management in an Object-Oriented Database System”, ACM SIGMOD Int. Conf. on Management of Data, Chicago, Illinois, Jun. 1988, pp. 37 - 45.

[Hadz91] Thanasis Hadzilacos and Vassos Hadzilacos, “Transaction Synchronization in Object Bases”, Journal of Computer and System Sciences, Vol. 43, No. 1, pp. 2 - 24.

[Herr90] U. Herrmann, P. Dadam, K. Kuspert, E. A. Roman and G. Schlageter, “A Lock Technique for Disjoint and Non-Disjoint Complex Objects”, 2nd Int. Conf. on Extending Data Base Technology, Venice, Italy, Mar. 1990, pp. 219 - 237.

[Jun95] Woochun Jun and Le Gruenwald, “Automation of Fine Concurrency in Object-Oriented Databases”, Proc. of the ISCA 8th Int. Conf. on Computer Applications in Industry and Engineering, Nov., 1995, pp. 72 - 75.

[Malt93] Carmelo Malta and Jose Martinez, “Automating Fine Concurrency Control in Object-Oriented Databases”, 9th IEEE Conf. on Data Engineering, Vienna, Austria, Apr. 1993, pp. 253- 260.

[Moss85] J. E. B. Moss, Nested Transactions : An Approach to Reliable Distributed Computing, MIT Press, Cambridge, 1985.

[Muth93] P. Muth, T. C. Rakow, G. Weikum, P. Brossler, and C. Hasse, “Semantic Concurrency Control in Object-Oriented Database Systems”, Proc. of the 9th IEEE Int. Conf. on Data Engineering, Apr. 1993, pp. 233 - 242.

[Ozsu91] M. T. Ozsu and Patrick Valduriez, Principles of Distributed Database Systems, Prentice Hall, 1991.

[Rese94] R. F. Resende, D. Agrawal, and A. E. Abbadi, “Semantic Locking in Object-Oriented Database Systems”, Proc. of OOPSLA 94, Portland, OR, USA, Oct. 1994, pp. 388 - 402.

[Serv90] Servio Logic Corp, “Chap. 16: Transaction and Concurrency Control”, in Gemstone Product Overview, Alameda, CA., 1990.