Semantic Locking in Object-Oriented Database Systems

R. F. Resende* D. Agrawal A. El Abbadi†
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

Object-oriented databases are being increasingly used to model non-standard applications that emphasize modularity, composition, and rapid prototyping. Semantic locking protocols are presented for transaction management for such object-oriented databases. In particular, the protocol incorporates the semantics of complex objects, nested executions and dynamic conflicts resulting from referentially shared objects.

Category: Research.

Topic Area: Datbases, Concurrent Systems.

1 Introduction

Object-oriented databases are being increasingly used to model non-standard applications. These applications place new constraints on databases design. In particular, they emphasize modularity, composition, and rapid prototyping. Often basic objects are defined, and more complex and advanced objects are composed recursively in terms of these basic objects. As a result, the traditional notion of conflict based on read and write operations may impose unnecessary constraints on the execution of transactions in such systems. To alleviate this problem, an appropriate approach is to use the semantics of higher level objects to ensure correct execution of transactions. Another aspect of an object-oriented database system is that an operation invoked on a higher level object results in an entire execution hierarchy on lower level objects, sometimes referred to as composite objects. If traditional approaches such as strict two phase locking are used to manage composite objects, the execution hierarchies of two conflicting operations must be completely disjoint from each other. This would be analogous to the approach of executing transactions serially in a shared database system. To remedy this problem, a protocol for executing transactions in object-oriented databases must

*Supported by CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico.
†Contact author: Email: amr@cs.ucsb.edu; Tel: (805)-893-4321; Fax: (805)-893-8553.

allow execution hierarchies of different transactions to be interleaved as long as their behavior is equivalent to an execution in which these hierarchies are disjoint. Finally, conflicts among execution hierarchies of transactions are often not known *a priori* rather they are established dynamically during the execution. Protocols for transaction management in object-oriented databases need to address all these issues, viz., higher level semantics, nested executions, and dynamic conflicts.

In relational databases, the data is passive and has a flat structure. Nested transactions have been proposed in that context but the motivation is mainly for modularity of application and program design. On the other hand, in an object oriented databases, objects are active and have a nested structure. As a result, nested executions is of fundamental concern for transaction management in object oriented databases. Furthermore, unlike relational databases where all interactions to the database is via read and write operations, objects have rich semantics associated with them. This semantics can potentially be used to increase concurrency among concurrent transactions. In this paper, we present locking protocols for transaction management in an object oriented databases. These protocols are novel since they incorporate the notion of semantics for executing nested transactions and are applicable for object oriented databases with referential sharing among disjoint objects. We believe this is an important property of object oriented systems since new objects are composed from existing objects, hence referential sharing among unrelated objects is unavoidable. Earlier work on transaction management in object oriented databases has either ignored the issue of semantics [HH91], or have only used semantics partially [AE92], or do not completely deal with referential sharing [MRW+93].

The paper is organized as follows. The model is described in Section 3. We motivate the notion of using semantics for object oriented databases and the protocol in Section 4. In Section 5, we describe the protocol that is based on the above design criterion. This protocol is extended for further concurrency in Section 5. We conclude with a discussion in Section 7.

2 Related Work

In traditional databases, the most commonly adopted correctness criterion is serializability. Object-oriented database systems such as ORION [GK88, Kim90] and O_2 [CF90, BDK92] ensure the serializable execution of transactions. Both of these database systems employ a locking protocol based on multigranularity locking [Gra78]. Multigranularity locking was developed to reduce locking overhead by organizing the database in the form of a hierarchy, e.g., the database is composed of files, and each file is composed of a set of records etc. By obtaining a lock at a higher level, transactions implicitly lock entities at the lower levels. The designers of ORION and O_2 developed similar protocols replacing the database hierarchy by the class and object hierarchy in object oriented databases. However, their approach is based on the traditional notion of *read* and *write* locks and the strict two phase locking protocol.

Thus, these approaches do not employ the semantics of higher level operations available in object-oriented databases. Furthermore if an object is locked, every descendant of that object in the hierarchy is also locked even if the transaction may only access a small subset of that sub-hierarchy. In particular, a significant drawback with this approach is the following. In object-oriented databases, a transaction execution begins by executing an operation or a method on a higher level object which in turn invokes operations or methods on other objects resulting in a nested structure of the transaction execution. By using the standard two-phase protocol with multigranularity locking, ORION and O_2 do not exploit the nested behavior of transaction executions to increase concurrency. However, the major strength of these two systems, from the transaction management point of view, is that they directly address the issue of schema evolution and extensibility. In particular, the proposed protocols allow concurrency between schema changes as well as user transactions. This approach was also applied to complex objects with pre-declaration of locks [HDK+90].

Moss [Mos85] introduced the notion of nested transactions for modular composition of applications and programs in database systems. He proposed the nested two phase locking protocol (N2PL) which is a generalization of the well-known two-phase locking protocol for concurrency control [EGLT76]. Beeri, Bernstein and Goodman [BBG89] presented a description of nested transactions systems, which includes a definition of serializability, and a set of basic techniques that can be used for proving the serializability of nested computations. Hadzilacos and Hadzilacos [HH91] simplified the model proposed by [BBG89] and applied it to transaction executions in object-oriented databases. In particular, Hadzilacos and Hadzilacos explicitly model the nested execution of transactions and propose a correctness criterion for such executions. They present a nested two phase locking protocol for an object oriented database that is a generalization of the locking protocol for nested transactions proposed by Moss [Mos85]. In this protocol, locks are requested for the atomic operations, which correspond to the leaves in the nested execution. When a method execution terminates, its parent inherits its locks. A lock is granted only if there are no other conflicting locks. Locks are discarded only when the top-level transaction terminates. The proposed mechanism, however, does not take advantage of the semantics of the methods at the higher levels.

Agrawal and El Abbadi [AE92] have proposed a concurrency control protocol for providing transactional access to object-oriented databases. They provided a uniform treatment for dealing with both class and instance objects and extend the two phase locking protocol to objects with arbitrary operations. The novel aspect of this protocol is that conflicting operations may share locks on the same objects, provided that the order between the operations is preserved. This protocol is extended for synchronizing nested executions of transactions on objects. Allowing updates to both classes and instances of these classes makes the proposed protocol beneficial for extensibility and experimentation, which are needed for continuously evolving object oriented database applications. However, this protocol also does not exploit the semantics of higher level methods to increase concurrency.

Muth et al. [MRW$^+$93] present a locking protocol for object oriented databases. The authors adopt a notion of commutativity similar to the one in [BBG89] in which the conflicts between lower level operations or methods can be ignored due to the commutativity of the corresponding higher level methods in the nested execution. However, the protocol cannot be used with object-oriented databases with referentially shared objects, i.e., non-disjoint complex objects. This is a severe limitation since composition is one of the fundamental properties of object-oriented databases as well as a necessary condition for modular design. Furthermore, such referential sharing cannot be described statically since it may arise dynamically depending upon the input parameters and the specific execution environment. This protocol requires locks not only for the atomic operations but also for method executions. In particular, the system designer has to derive a conflict or commutativity table not only for the atomic operations of each class but also for all the method executions of all the classes in the database.

We have developed a locking-based concurrency control protocol for object-oriented databases. Our protocol generalizes Moss's original nested transaction protocol in that it incorporates the rich semantic information available in object oriented databases. Our protocol incorporates the notions of semantics of higher level objects [BBG89, MRW$^+$93], nested executions [HH91, AE92, MRW$^+$93] and dynamic conflicts. In particular, we address the issue of referentially shared objects where the referential sharing is determined dynamically during the execution. The correctness of our protocol is based on the theory of nested transactions proposed in [BBG89].

3 The Model

An *object oriented database* is a collection of *classes* and *instances* of these classes. A class defines a set of attributes for its instances and methods or procedures that are used to manipulate these instances. We assume that each class definition is physically represented as a *class object* and each instance is represented as an *instance object*. Thus, we use the generic term *object* to refer to both classes and instances of these classes. An object supports both atomic operations on its local state as well as methods to access both the local state as well as to access other objects. Users access objects by executing methods defined on objects or atomic operations associated with objects. A method execution is a partial order of method executions and atomic operations [HH91]. Such a caller-callee relationship between methods establishes a hierarchy that is known as *nested transactions* or *nested method executions* [Mos85]. We assume that the objects are ordered in a hierarchy and a method in one object can only call methods on objects that are lower in the hierarchy.

Figure 1 shows a database schema consisting of three classes of objects: Customer, Item, and Order. This example is similar to the one in [MRW$^+$93] but is extended to a more

class Customer
customerNo: integer
name: string
orders: **set**(Order)

class Item
itemNo: integer
name: string
quantity: integer
orders: **set**(Order)
price: float

class Order
orderNo: integer
customer: Customer
status: integer
itemset: **set**(**tuple**(item: Item,
quantity: integer,
price: float))

Figure 1: Attributes of classes Customer, Item, and Order

realistic setting that supports referentially shared objects. Each customer can have several orders and each order may mention one or more items. Furthermore, each item could have been mentioned by several orders. Note that a particular object of class Order could be referentially shared among a customer and several items.

Figure 2 illustrates the skeleton of two methods. The class Customer defines a method "status-of-orders()" which invokes method "report-status()" on each of the orders of the customer where the status of an order indicates whether the order is pending, processed, etc. The class Order computes the total value of the order by simply accessing the quantity and price of each tuple in the set items. There is a provision for the price of some item to be different from the list price of that item, perhaps due to a discount. The method "total-list-price()" defined in the class Order computes the list-price for the order. In the class Order, the method "report-status()" is implemented with atomic operation "read(status)". Similarly in class Item, "report-price()" is implemented with atomic operation "read(price)"
.

As in [HH91], we assume that user transactions are methods of a distinct object called *environment*. We will be using the terms transaction and method execution interchangeably. We also assume the existence of a special method execution, T_0, in the environment object that invokes the top-level method executions. Figure 3 shows an example of transactions causing nested calls of method executions in an object oriented database whose objects are instances of the classes in Figure 1. The execution order is from left to right. Method executions are represented by circles, and atomic operations are represented by squares. Transaction T_1 calls method "status-of-orders()" on Customer object John. The transaction T_1 is invoked with a key value such as a name customerNo and uses the select operation to retrieve the object-id of John. For the sake of simplicity, we omit such operations. John has two orders o1 and o2. Transaction T_2 computes the total list price of two orders o2 and o3. Order o2 is for two items gadget1 and gadget2 and o3 is for gadget2. Since there is one

```
method status-of-orders():
    ...
    for each orders objectID do
        ...
        objectID:report-status()
        ...
    od
    ...
end method;
```

```
method total-list-price():
    ...
    for each itemID in itemset do
        ...
        read(itemID.quantity )
        ...
        itemID.item:report-price()
        ...
    od
    ...
end method;
```

Figure 2: Examples of Methods

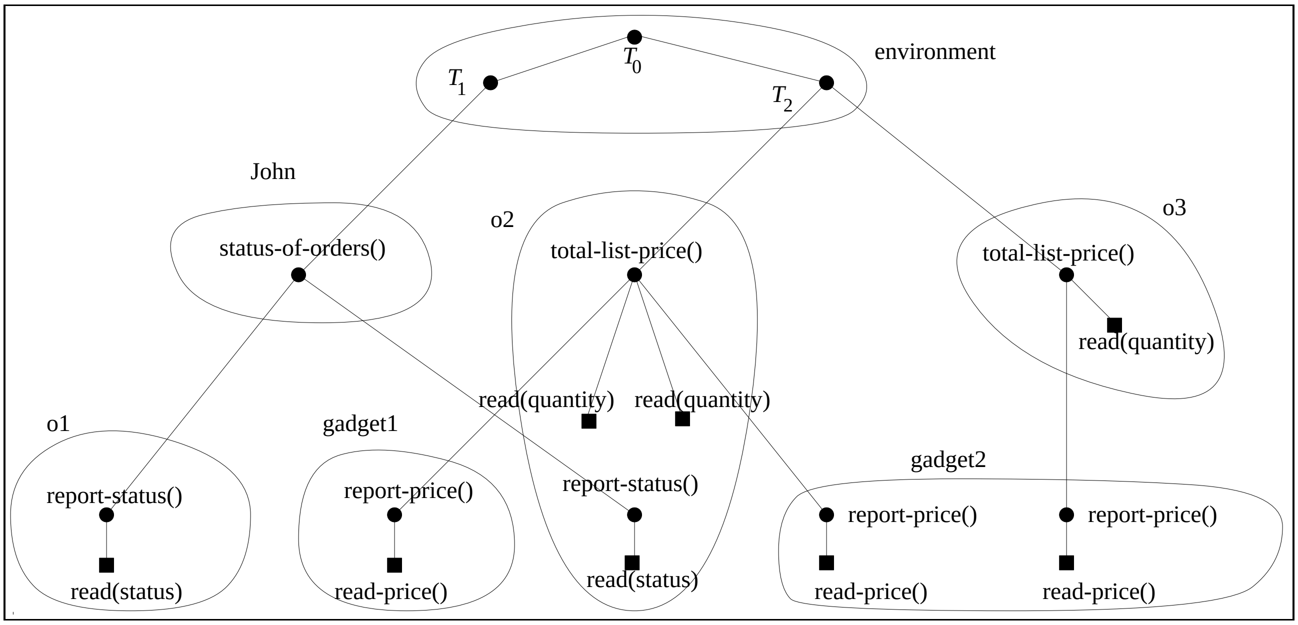


Figure 3: Concurrent execution of two transactions

customer for each order, from the figure we can conclude that John is the customer of order o2 and has ordered gadget1 and gadget2.

Standard tree terminology is used to refer to relationships among method executions. For example method executions "total-list-price()" on objects o2 and o3 are *children* of T_2 in Figure 3. Node x is a *descendent* of y if $x = y$ or x is a child of a descendent of y. Node x is a proper descendent of y if it is a descendent other than y itself. *Ancestor* relationship is defined analogously. Two method invocations x and y on the same object commute if and only if the two possible sequential executions of x and y are indistinguishable for both x and y and for all possible sequences of methods that may be invoked subsequently [CF90, HW91, MRW+93]. Table 1 illustrates the commutativity relationships between the methods of class object *Item* (note that the entry $\perp$ indicates that the relationship is undefined).

decrease-percent()	report-price()	change-price()	
commute	$\perp$	$\perp$	decrease-percent()
$\perp$	$\perp$	$\perp$	report-price()
$\perp$	$\perp$	$\perp$	change-price()

Table 1: Commutativity Table for Object Class *Item*

We assume that the commutativity relationships between the operations are well-defined and can be derived based on the semantics and the specifications of the class and its methods. However, in a dynamic object oriented database system, methods on different objects may invoke methods on the same objects resulting in conflicts (specially when referential sharing is allowed). Such a conflict cannot be determined a priori based on the static analysis rather it materializes as the particular methods are executed. N2PL captures these conflicts using lock inheritance, however, it fails to take advantage of any object semantics. Our approach extends N2PL to object-oriented databases and uses the additional semantic information to increase concurrency.

Beeri, Bernstein, and Goodman [BBG89] developed a general theory to reason about nested executions with semantics. They introduced two transformations that establish an equivalence relationship between executions: *substitution* and *commutativity-based reversals*. *Substitution* can be specialized in terms of reduction and expansion. Using sequences of these two transformations we can establish the equivalence of some execution and its serial counterpart, proving serializability of nested executions. A transaction x is *separated* if no other operations of other transactions are interleaved with its leaves. Consider a transaction x where all its children are leaves and it is separated. *Reduction* is a transformation that relates two nested executions where the first one has a separated transaction x whose children are leaves and the second is the same as the first except that x was substituted by the atomic operation corresponding to transaction x. (The underlying assumption is the existence of a correspondence between a transaction and an atomic operation, i.e., if the execution of a transaction is atomic then there exists a corresponding atomic operation.) *Expansion* is the inverse of reduction. The transformation *commutativity-based reversal* relates two computations where the first has two consecutive leaves that commute and the second is the same as the first except that these two consecutive leaves are reversed. We call conflict serializability the notion of serializability where all conflicts are preserved, and semantic serializability the serializability where some conflicts can be ignored based on the semantics of operations at higher level [BHG87, BBG89, MRW+93].

4 Motivation

Locking is a mechanism commonly used to solve the problem of synchronizing access to shared resources. *Locks* are place holders for checking concurrency control predicates. Each atomic operation has a lock associated with it. Before a method execution may execute an atomic operation, a lock must be requested and granted. We say that a method that executes an atomic operation x *holds* lock $lock(x)$. We say that $lock(x)$ *conflicts with* $lock(t)$, if atomic operation x conflicts with atomic operation t. When a method execution t terminates, its parent *inherits* all the locks t owns. A special case occurs when the terminating method execution is a top-level one. In this case the locks are released, i.e. they cease to exist. The locks that a method execution x inherit from method executions that are its children are said to be *retained* by x. This means that there are two ways for a method execution to own a lock. Method execution x' can own $lock(x)$ in the sense of holding, meaning that atomic operation x is executed directly by x'. Method execution x' can own $lock(x)$ in the sense of retaining, meaning that atomic operation x was executed by some proper descendent of x'. In N2PL $lock(x)$ can be granted to method execution x', i.e. atomic operation x can be executed by x', iff no other method execution holds a conflicting lock, and all the retainers of a conflicting lock are ancestors of x'. We summarize the nested two-phase locking (N2PL) rules adapted from [Mos85] as follows.

1. A method execution t' can execute an atomic operation t, iff $lock(t)$ is requested and is granted (we say that the method execution t' *holds* $lock(t)$).
2. A method execution cannot terminate (commit or abort) until all its children have terminated. When a method execution terminates:
   (a) *It is not top-level and it commits:* Its locks are inherited by its parent (we say that the parent *retains* the inherited locks).
   (b) *It is not top-level and it aborts:* Its locks are discarded.
   (c) *It is top-level:* Its locks are discarded.
3. A lock $lock(x)$ can be granted to a method execution iff

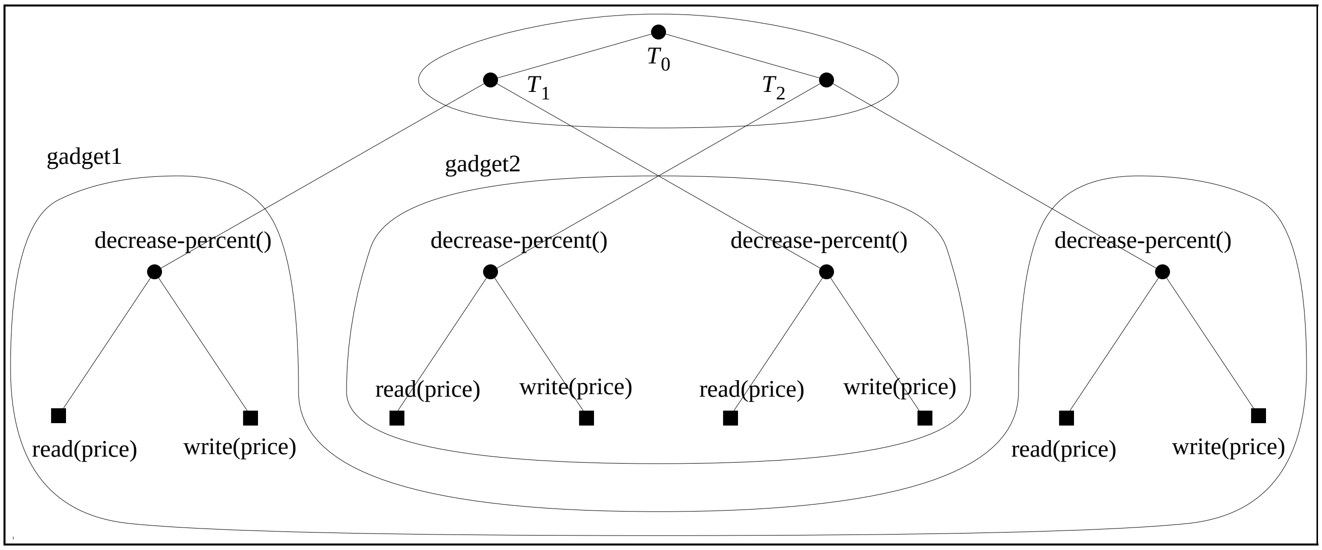


Figure 4: A semantic serializable execution

(a) No other method execution holds a conflicting lock, and

(b) All the retainers of a conflicting lock are ancestors of the requesting method execution.

It is shown in [BBG89, HH91] that N2PL ensures serializability. N2PL does not take advantage of any knowledge about the semantics of the methods being executed. We now show an example of how N2PL can potentially take advantage of the semantics of method executions in order to ignore certain conflicts and accept more executions.

Suppose that the class Item of Figure 1 has a method "decrease-percent()". This method decreases the price of its corresponding item. Observe that two instances of method "decrease-percent()" commute since percentage operations commute. Figure 4 shows two transactions T_1 and T_2 that execute methods "decrease-percent()" concurrently on gadget1 and gadget2. Observe that method "decrease-percent()" is implemented using two atomic operations "read(price)" and "write(price)". In the traditional conflict serializability that does not take advantage of semantics of higher level operations in nested transactions, this execution would be considered non-serializable. In this execution, transaction T_1 must be serialized before transaction T_2 because of the conflicts in gadget1 and transaction T_2 must be serialized before transaction T_1 because of the conflicts in gadget2. However, if the semantics of "decrease-percent()" is taken into consideration (see Table 1), two such operations commute and hence their relative order is not significant. Therefore, this execution will be serializable and is equivalent to both T_1 before T_2 or vice-versa. Note that there are cases where semantic information cannot be used to ignore conflicts at the lower level. For example, consider

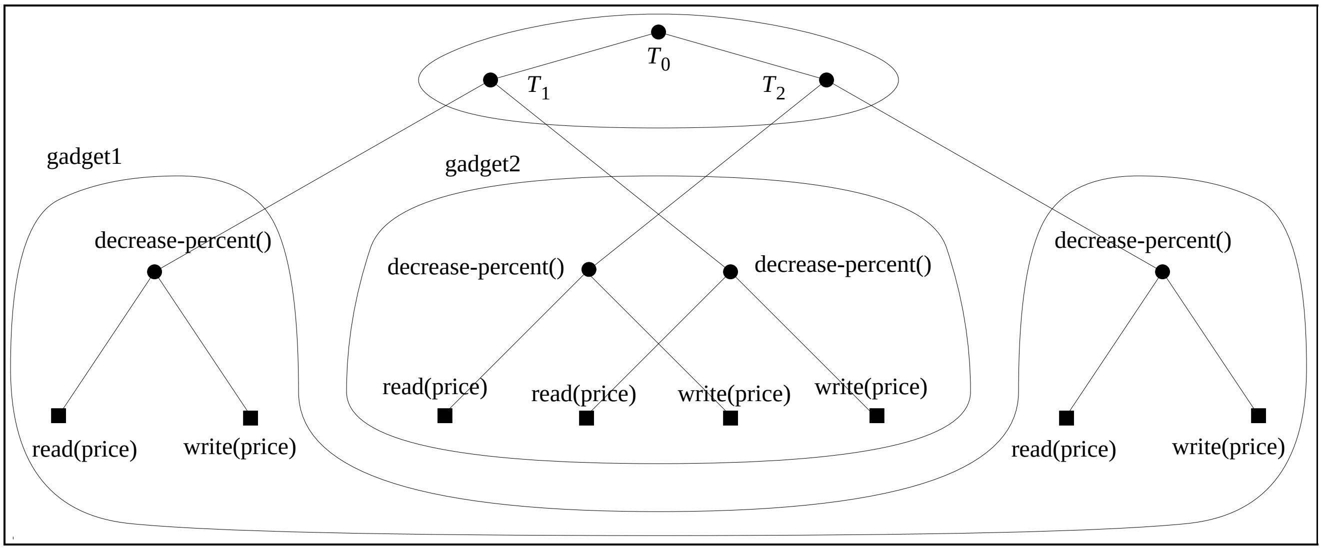


Figure 5: A nonserializable execution

another execution of the same two transactions that is illustrated in Figure 5. In this execution we cannot apply the commutativity information between the two "decrease-percent()" method executions on gadget2. This is because the commutativity of these two method executions is derived assuming the atomicity of the operations. In this execution the two method executions are not executed atomically with respect to each other.

Using the theory described in [BBG89] we can establish the semantic serializability of the execution showed in Figure 4. We can *reduce* the computations rooted at method executions "decrease-percent()" since these computations are separated. Now we can use the semantics of the Item object to apply *commutativity-based reversals* on the reduced method executions "decrease-percent()" of gadget2. Basically nested two phase locking does not have any mechanism that allows lower level conflicts (such as "read(price)", "write(price)" in Figure 4) to be ignored by using the semantics at higher level methods (e.g., "decrease-percent()"). In the next section we present semantic locking, a protocol that uses semantics to increase concurrency in object oriented databases.

5 Semantic Locking

We start by describing some of the salient aspects of object-oriented databases that are considered in the design of our protocol. In general, in object-oriented databases, the conflicts and commutativity relations between methods are defined on a per class basis. Methods on different objects are generally assumed to commute with each other [GK88, CF90, HH91, MRW+93]. This approach is reasonable for object-oriented databases with disjoint complex objects. However, in a system with referentially shared sub-objects, the commutativity relation needs to be redefined even for methods that are executed on different objects. One approach could be to define these relations across objects statically but this approach could be very conservative and defeats the goal of modularity and extensibility in object-oriented databases. Furthermore, the referential sharing of objects may occur dynamically during execution. We assume that commutativity relations are defined for atomic operations. This is the same assumption made by most previous work in both nested transactions and object oriented databases [Mos85, HH91, FLW90]. However, we enrich this model by exploiting semantic information, i.e., commutativity relations that are derived from the semantics of methods of classes. The conflict and commutativity relations between methods that referentially share sub-objects are left unspecified and are determined dynamically during the execution of such methods. Thus, tests for conflicts are done only for methods executing at the same object. Methods in different objects will execute concurrently and depending on their executions, the proposed protocol will capture the conflict relations among such methods.

5.1 The protocol

In semantic locking each atomic operation has a *lock* associated with it. Before a method execution may execute an atomic operation, a lock must be requested and granted. This is similar to [Mos85], locks are only required for the execution of atomic operations and contrasts with [MRW+93] where locks are required for the execution of methods. Similar to [Mos85] when a method execution terminates its parent inherits all its locks. The locks that a method execution x inherits from its children are said to be *retained* by x. In semantic locking, locks can be granted to conflicting operations if the corresponding operations have commuting ancestors. The important observation to make regarding semantic information is that two methods, which are known to commute, commute only if both are executed atomically . Hence in Figure 4 a pair of "decrease-percent()" commute if the operations implementing these methods do not interleave. Once all the operations implementing one "decrease-percent()" are executed the operations of another "decrease-percent()" can execute. We implement this idea by allowing "decrease-percent()" of T_1 to acquire the lock for "read(price)" only after the locks on "read(price)" and "write(price)" are inherited by T_2. In particular, a lock request $lock(t)$ which conflicts with another lock, $lock(x)$ can be granted using semantic information only when $lock(x)$ is inherited by a node whose descendent commutes with an ancestor of $lock(t)$. In this case, we say that the pair of commuting ancestors *relieves* the conflict among atomic operations. More formally assume that $lock(t)$ is requested and it conflicts with $lock(x)$. If $lock(t)$ is granted while $lock(x)$ is still retained by some method execution because there are commuting ancestors x' and t' of nodes x and t respectively then we say that (x', t') *relieves* conflict (x, t).

We now describe the protocol which is referred to as *semantic locking*. We introduce the following notation for intervals of related nodes to simplify the presentation of the protocol. Let x be a method execution and let x' be a descendent of x, $[x \cdots x']$ represents the set $\{a \mid a \in descendents(x) \wedge a \in ancestors(x')\}$. If we substitute '[' or ']' by '(' or ')' the proper descendents/ancestors should be taken e.g. $(x \cdots x'] = [x \cdots x'] - \{x\}$. A method execution must observe the following semantic locking rules:

1. A method execution t' can execute an atomic operation t, iff $lock(t)$ is requested and is granted (we say that the method execution t' *holds* $lock(t)$).
2. A method execution cannot terminate (commit or abort) until all its children have terminated. When a method execution terminates:
   (a) *It is not top-level and it commits:* Its locks are inherited by its parent (we say that the parent *retains* the inherited locks).
   (b) *It is not top-level and it aborts:* Its locks are discarded.
   (c) *It is top-level:* Its locks are discarded.
3. A lock $lock(t)$ can be granted to a method execution iff
   (a) No other method execution holds a conflicting lock, and
   (b) For all other non-ancestor methods x'' that *retain* a conflicting $lock(x)$, some element x' of $(x'' \cdots x]$ and some ancestor t' of t commute ((x', t') relieves the conflict (x, t)).

According to Rule 1, locks must be acquired only for the leaves (atomic operations). Rule 2 enforces the strict two phase locking restriction for nested executions. Rule 3 relaxes the nested two phase locking constraint by using a possibly available commutativity relation between method executions based on semantic information. In nested two phase locking, a lock can be acquired provided:

- No other method execution holds or retains a conflicting lock; or
- if there are conflicting locks, such locks are retained by the ancestors.

Semantic locking grants a lock in the above cases as well as in the following case:

- If there are conflicting locks retained by non-ancestors then one of the ancestors of the operation up to but not including its retainer and some ancestor of the requester commute.

In this latter case semantic information is used by the locking protocol, hence the name semantic locking protocol.

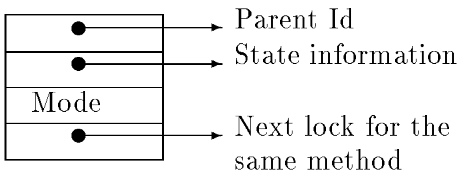
5.2 Implementation issues

We briefly discuss an implementation of the semantic locking protocol proposed in this paper. Our approach is to generalize the implementation proposed for nested two-phase locking protocol [Mos85] and we have chosen a similar terminology to that in [MRW+93]. Obviously semantic locking demands more implementation effort than N2PL, but this extra demand does not add significant complexity. Objects have methods and atomic operations, and belong to a certain object class. The scheduler maintains for each object class:

- A conflict table indexed by the atomic operations. An entry in a conflict table indicates whether any two atomic operations conflict or not;
- A commutativity table indexed by methods. An entry in a commutativity table indicates that it is unknown that the methods commute or whether they commute.

The scheduler also maintains a lock table in each object that determines which atomic operations have locks requested and granted. The following structure adapted from [Mos85] illustrates some of the relevant fields of an entry in a lock table:

Lock Record Format



The lock record entry *parent-id* identifies the transaction who currently holds/retains the lock on this object. The entry *state-information* is necessary for recovery purposes which is not dealt with in this paper. The entry *mode* indicates the type or name of the atomic operation being executed on this object. The last entry is an additional information that is needed primarily for semantic locking. In particular, the entry points to a list of ancestors of the transaction that initiated the atomic operation. This information will be used by the scheduler to determine if a certain conflict at the atomic operation level can be relieved due to commuting ancestors (Rule 3(b) in Section 4.1).

Now we describe the scheduling actions enforced in the semantic locking protocol. Methods can be called at any time and the scheduler keeps track of the hierarchy. When an atomic operation is invoked a lock must be obtained before the operation is executed. We will refer to the parent of the operation as *the requester*. The scheduler uses the conflict and lock tables to check if a lock can be granted. If the requested lock does not conflict with other locks or if there is a conflict and the corresponding operations have the same parent, an entry is created in the lock table and the lock is granted, allowing the operation to execute. Otherwise, if the lock conflicts with other locks whose operations are not siblings, an entry is created in the lock table but the granting will have to wait for the termination of some method executions as follows. We must find for each operation corresponding to a conflicting lock the lowest ancestor that commutes with any ancestor of the requester (or the corresponding child of their least common ancestor). In this case, we use the commutativity table to determine whether the methods commute. Assume that a lock was requested for operation p, and that $p.conflict\_set$ denotes the set of operations that conflict with p, $p.waits\_for\_set$ denotes the set of methods that p has to wait for before its lock can be granted. Let $p.object$ the object where p was invoked. Figure 6 illustrates the steps involved in acquiring locks. In 2PL, the wait-for-set of a conflicting operations consists of the top-level transactions that hold conflicting locks. In N2PL, this set consists of the least common ancestors of the holders/retainers of the conflicting lock and the lock requester. In our case, on the other hand, the set may contain descedants of the least common ancestors if these descendants commute with an ancestor of the requester.

---

for "each q in p.conflict_set" **do**
    add to p.waits_for_set the lowest commuting ancestor
    of q with respect to the conflict between q and p,
    if they do not have commuting ancestors add the child of
    least common ancestor of p and q that is an ancestor of q;
od
install an entry in lock table of p.object for $lock(p)$ as requested;
wait until the nodes in p.waits_for_set terminate;
modify the entry in lock table for $lock(p)$ as granted;
grant $lock(p)$ to the requester;

Figure 6: A pseudo-code for acquiring locks

When a method execution terminates the scheduler signals the operations that are waiting. When a top-level transaction terminates all its locks are released and the information about the operations of this transaction can be thrown away. One way to implement this is to maintain a reference to all objects that were visited by descendents of method executions. If the information about commutativity of methods is not used our implementation is essentially the same implementation as in N2PL [Mos85].

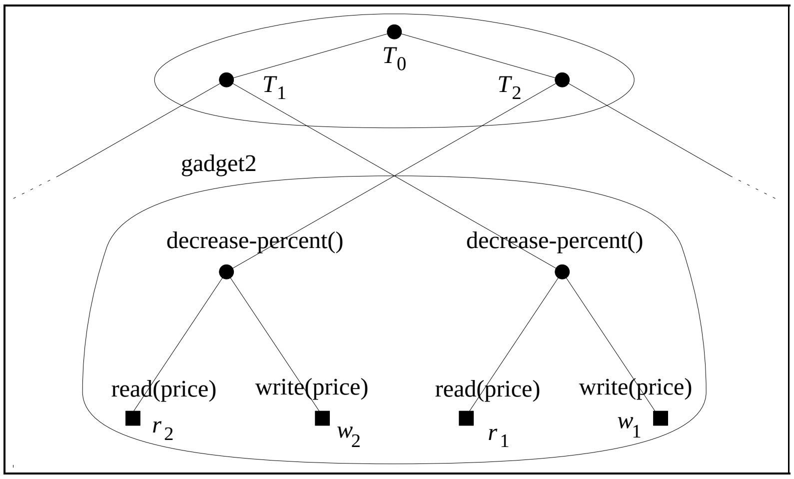


Figure 7: How locks are granted in gadget1.

We use the example in Figure 7, to illustrate the dynamics of our design. Method execution T_2 of the environment object is invoked and calls method execution T_2.decrease-percent() of object gadget2. Method execution T_2.decrease-percent() will need to acquire locks on atomic operations r_2 and w_2. Operation w_2 conflicts with r_2 but both have the same requester. Hence, the lock corresponding to w_2 can be granted to T_2.decrease-percent(). Now, the lock table of gadget2 has two entries corresponding to r_2 and w_2. T_1 calls method execution T_1.decrease-percent() of gadget2, but $lock(r_1)$ cannot be immediately granted since r_1 conflicts with w_2. The commutativity table of class Item indicates that decrease-percent() methods commute with each other, hence T_1.decrease-percent() in gadget2 waits until T_2.decrease-percent() in gadget2 terminates. (T_2.decrease-percent() is the lowest ancestor of w_2 that commutes with some ancestor of r_1, in this case T_1.decrease-percent()). After T_2.decrease-percent() terminates and $lock(r_1)$ is granted the lock table of gadget2 has three entries corresponding to r_2, w_2 and r_1. Similarly $lock(w_1)$ cannot be immediately granted since w_1 conflicts with r_2 and w_2 (it also conflicts with r_1 but they have the same requester). The lowest ancestors of r_2 and w_2 that commute with an ancestor of w_2 are the same node i.e. T_2.decrease-percent(). Since it is terminated, $lock(w_1)$ is granted. Observe that if there are no ancestors that commute, e.g., decrease-percent() methods in the example, a conflicting node will have to wait for the termination of the corresponding top-level transaction.

5.3 Analysis

We underscore the distinction between nested two phase locking and semantic locking with the help of an example. In particular, recall the execution illustrated in Figure 4. Nested two phase locking cannot derive this execution. In particular, T_1 is blocked by T_2 at the level of gadget2. Method execution "decrease-percent()" of transaction T_1 is blocked trying to acquire a lock on method "read(price)" (it conflicts with the lock on method "write(price)" held by "decrease-percent()" of transaction T_2). Similarly, T_2 is blocked by T_1 at the level of gadget1. Method execution "decrease-percent()" of transaction T_2 is blocked trying to acquire a lock on method "read(price)" and hence deadlocked. When semantic locking is used for this execution there will be no deadlock. Consider for example the method executions of T_1 and T_2 on the Item gadget2. When T_1 executes its "read(price)" on gadget2, it conflicts with the "write(price)" executed by T_2. However, the semantic locking rule permits "decrease-percent()" of T_1 to acquire a lock for executing "read(price)" on gadget2 after "decrease-percent()" of T_2 terminates and the lock corresponding "write(price)" on gadget2 is retained by T_2. Since commutativity of "decrease-percent()" is defined for completed operations, the "decrease-percent()" executed by T_1 must wait until the "decrease-percent()" executed by T_2 has completed for the commutativity property to hold. This explains the reason for excluding the retainer x in the expression $(x \cdots x']$ of Rule 3 in the protocol. However, note that even with semantic locking, deadlocks are possible. For example, in Figure 5, semantic locking results in a deadlock. This occurs since both T_1 and T_2 are both blocked when trying to obtain locks on gadget2.

To reason about the correctness of semantic locking we use the two transformations described in [BBG89]. We illustrate the use of these two transformations in Figure 8 using the example of Figure 4. Since each "decrease-percent()" in execution (a) is separated we can reduce them. We can go from execution (a) to execution (b) by using four reductions. Even though we maintain the labels "decrease-percent()", in execution (b) these nodes are actually substituted by the corresponding atomic operations. In execution (b) we can apply commutativity-based reversal in order to separate T_1, coincidentally we also separate T_2 and we obtain execution (c). In execution (c), since T_1 and T_2 are separated we can use two reductions to obtain execution (d) which is a serial execution.

We now use the bottom-up proof paradigm of [BBG89] to informally argue about the correctness of semantic locking. The goal is to create a sequence of equivalent executions starting from an execution, C_0, resulting from semantic locking and terminating with a serial execution, C_{serial}, consisting of top-level transactions. The proof paradigm is to establish a sequence of equivalent executions and the intermediate executions in this sequence are referred to as C_i. We choose a node, separate it using as many commutativity-based reversals as necessary, and then reduce it. We repeat this separation and reduction process until only the top-level method executions are left. With this we establish the equivalence of the execution produced by semantic locking with a serial execution consisting of only the operations corresponding to the top-level method executions.

In each iteration we choose for the separation and reduction the node x whose rightmost

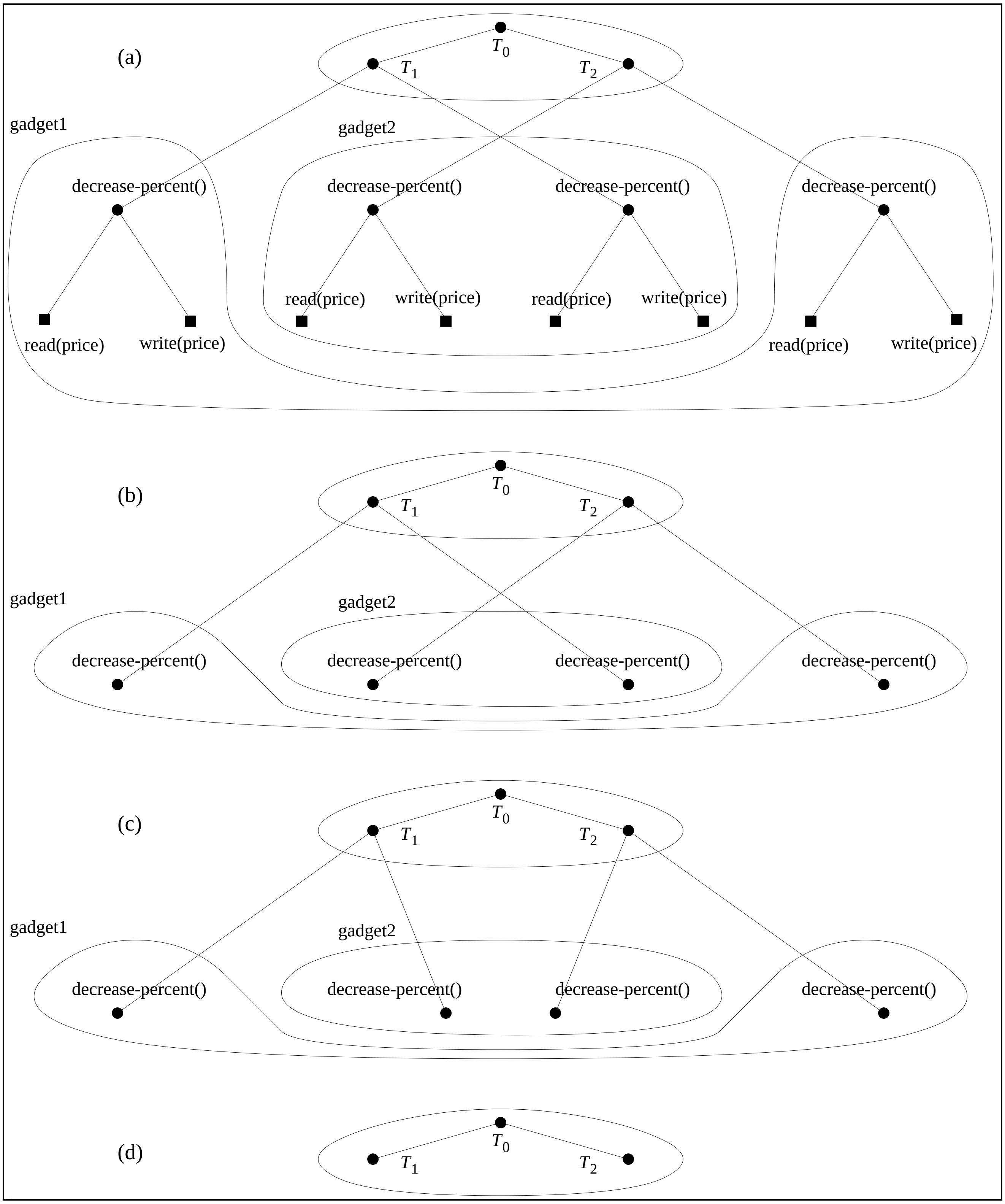


Figure 8: Using commutativity-based reversals and reduction

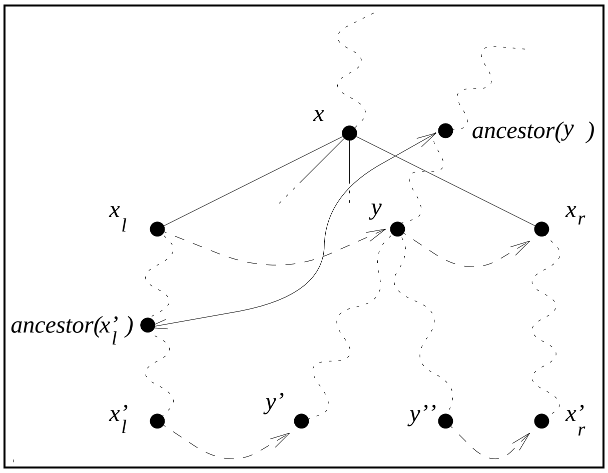


Figure 9: A hypothetical node y blocks x's separation.

child is the leftmost among nodes that have only leaves as children. We have to prove that it is always possible to separate x. We say that an operation *interleaves* within x if there are children of x at the left as well as the right of this operation. Our proof obligation is to show that in spite of such interleavings, there always exists such an x that can be separated. For C_0 no operation t that conflicts with a child of x interleaves within x. This is the case since from rule 3 of the protocol t must wait at least until x has terminated. We now need to argue that this separation and reduction can be performed when we are dealing with any intermediate execution C_i.

For the sake of contradiction assume that in C_i we cannot separate x whose rightmost child is the leftmost among nodes that have only leaves as children. This may be because x_l, a child of x, conflicts with and precedes y and that y conflicts with and precedes x_r, another child of x as shown in Figure 9. Since x_l conflicts with y there exist descendents x'_l and y' of x_l and y respectively in C_0, such that x' conflicts with and precedes y'. Assume that this conflict was not relieved. Then y' must have waited until the lock on x'_l was released or inherited by an ancestor of y'. In both cases, and due to the choice of x, all descendants of x must have terminated before y' was executed. Hence, y could not have been interleaved within the children of x. Next assume the conflict was relieved by a pair: ancestor of x'_l, $ancestor(x'_l)$, and a proper ancestor of y, $ancestor(y)$, as shown in the figure. Since commutativity is defined on a per object basis, both $ancestor(x'_l)$ and $ancestor(y)$ must be methods of the same object. Since $ancestor(x'_l)$ is an ancestor of x'_l and $ancestor(y)$ is a proper ancestor of y, then by our assumption of hierarchy of objects, x_l must be a method executed on an object higher in the object hierarchy than the object on which y was executed. A similar argument can be used to show that if y conflicts and precedes x_r, y is a method execution higher in the object hierarchy than method execution x. Since x is a parent of x_l this violates our assumption that method executions are always called from a higher level to a lower level in the object hierarchy. This argument can be extended to the general case when a sequence of nodes $y_1, \ldots, y_n$ are interleaved within x [Res94].

We conclude this section by noting that the protocol depends crucially on the assumption that objects are modularly designed and that they are hierarchically organized in a static manner. Although this is a desirable property, in practice, object oriented databases may be designed in such a way as to violate this restriction. In this case, Rule 3 of the protocol needs to be modified so as to dynamically capture any violations of this ordering in a particular execution. For example, a pair (p,q) of commuting methods may not be used to relieve a conflict if another pair (x, t), where q is a proper descendant of t and p is a proper ancestor of x, was already used to relieve a conflict. In general, this restriction must be enforced not only in a pairwise manner rather transitively in an execution. By enforcing this restriction dynamically we can ensure that no incorrect executions occur without imposing a static hierarchy on the objects. The details of the implementation and correctness proof of this approach appear in [Res94].

6 Semantic Locking with Ordered Shared Locks

Semantic locking allows more concurrency when compared to nested two phase locking. However, it may cause unnecessary blocking of concurrent method executions. If two operations conflict then one of them will have to wait until the lock of the other is released or that the execution reaches a level where the conflict can be relieved. This is mainly due to the fact that the traditional locks are limited to two types: *shared* and *non-shared* locks. This results in the unnecessary blocking of conflicting method executions. Recently, Agrawal and El Abbadi [AE90] have introduced a new type of relationship between locks called *ordered sharing*. Ordered sharing permits concurrent execution of conflicting operations with some constraints on termination. Ordered sharing has been adapted to nested two phase locking in object-oriented databases [AE92]. We now extend semantic locking to incorporate locks with ordered sharing.

When ordered shared relationships between locks are used, a lock request is always granted: either it is granted because there are no other locks of conflicting operations or if there are such locks then the lock is granted and an ordered shared relationship is established from all such locks. Informally, ordered sharing enables an operation t that conflicts with another operation x to start executing before $lock(x)$ is released. If a method execution t' owns locks that have an ordered shared relationship with respect to locks owned by x', constraints are imposed on the termination of t' to ensure correctness. We now describe the semantic locking protocol with ordered sharing.

1. A method execution t' can execute an atomic operation t, iff $lock(t)$ is requested and is granted (we say that the method execution t' *holds* $lock(t)$).

2. A method execution cannot terminate (commit or abort) until all its children have terminated. When a method execution terminates:

   (a) *It is not top-level and it commits:* Its locks are inherited by its parent (we say that the parent *retains* the inherited locks).

   (b) *It is not top-level and it aborts:* Its locks are discarded.

   (c) *It is top-level:* Its locks are discarded.

3. A lock $lock(t)$ can be granted to a method only if

   (a) No conflicting locks are being *held* and for all other non-ancestor method execution x that *retains* a conflicting $lock(x')$, some element of $(x \cdots x']$ and some ancestor t' of t commute ((x', t') relieves the conflict (x, t)); or

   (b) $lock(t)$ will be acquired with an ordered shared relationship with respect to a conflicting $lock(x')$ and either

      i. $lock(x')$ is being *held* by $parent(x')$; or

      ii. $lock(x')$ is *retained* by a method execution x with no element in $(x \cdots x']$ commuting with some ancestor of t.

   In this case, $lock(x')$ has an ordered shared relationship with respect to $lock(x')$, and we say that the ordered shared relationship is from $lock(x')$ to $lock(t)$.

4. An ordered shared relationship from $lock(x')$ to $lock(t)$ is removed when $lock(x')$ is retained by x and some ancestor in $(x \cdots x']$ commutes with some ancestor of t.

5. *Ordered Commitment Rule.* Method execution t is said to be waiting for x if t owns a lock with an ordered shared relationship with respect to a lock owned by x and x is a proper descendent of $parent(t)$. In this case t cannot commit as long as any method execution x, for which t is waiting for has not terminated.

Rules 1 and 2 are identical to the protocol presented in Section 5. The main distinction between the two protocols is as follows. In semantic locking, a lock request cannot be granted until either the lock is released or the computation reaches a point where method executions commute. On the other hand, in semantic locking with ordered sharing, a lock is always granted perhaps with ordered shared relationship. However, this relationship is dropped when the computation reaches a point where method executions commute. If the ordered shared relationship does not disappear, it imposes certain constraints on the terminations of conflicting ancestors in the computations as described in Rule 5. Rule 5 guarantees that when there is an ordered shared relationship from $lock(x)$ to $lock(y)$ this imposes an order on the termination of the two ancestors of x and y that are siblings. This is to ensure the

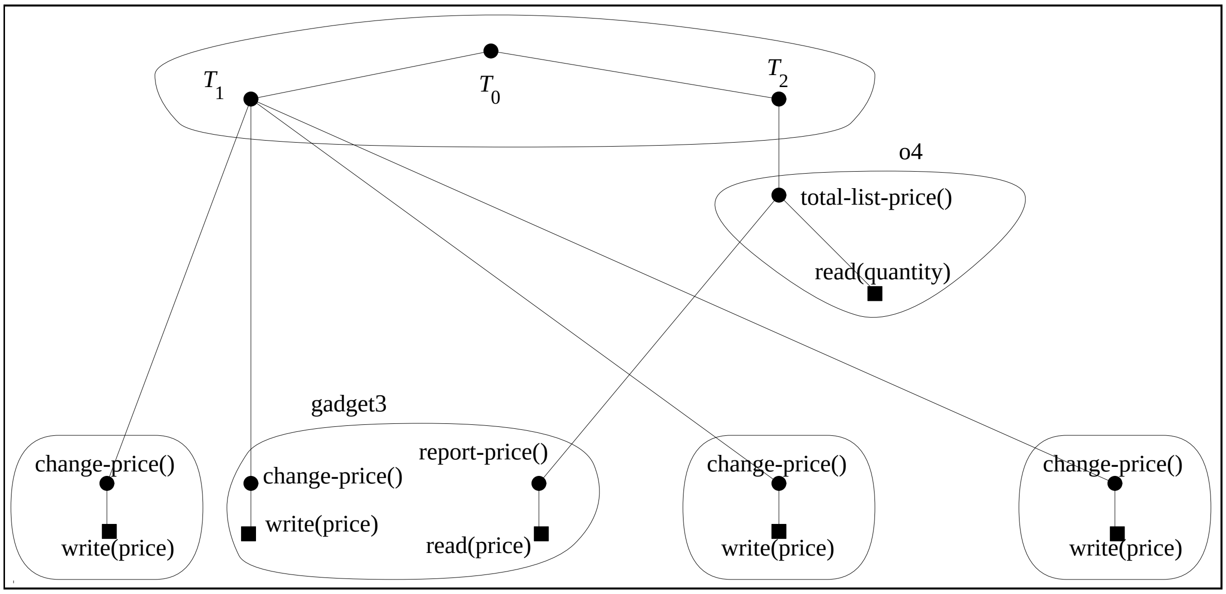


Figure 10: Use of ordered shared locks

serializability of the nested computations rooted at ancestors of x and y that are siblings [AE92, Res94].

Figure 10 shows an example of an execution of two transactions that use ordered shared locks. Transaction T_1 updates the price list of four items. Class Item defines a method "change-price()" for such updates. T_2 is another transaction that computes "total-price-list()" for only a single order: o4. Consider the situation when T_1 retains a lock on operation "write-price()" of gadget3, which is the only item of order o4. When method "report-price()" with ancestors "total-list-price()" on order o4 and T_2 requests a lock on operation "read(price)" of gadget3, the semantic locking protocol would not grant the lock. This happens because "write-price()" conflicts with "read-price()" and there are no pairs of ancestors that eliminate the conflict. Hence, the execution shown in Figure 10 will not be accepted by semantic locking. However, when semantic locking with ordered sharing is used, the lock request of "report-price()" is granted creating an ordered shared relationship from "write-price()" to "read-price()" on gadget3. As a result, the execution shown in Figure 10 will be accepted by semantic locking with ordered sharing. Note that due to the ordered shared relationship transaction T_2 cannot commit until transaction T_1 terminates.

7 Conclusion

In this paper, we presented semantic locking protocols for transaction management in object-oriented databases. Unlike traditional locking protocols, our approach serializes transactions at the level of method executions, i.e., the protocols take into consideration the semantics of methods executions. In addition, our protocols are novel in the sense that they support referentially shared objects as well as conflicts that arise dynamically during execution. We first presented the semantic locking protocol which is a generalization of nested two phase locking in object oriented databases [Mos85, HH91, MRW+93]. Although this protocol provides more concurrency than the traditional approach it still may unnecessarily delay execution of some methods. We remedy this problem by incorporating a new type of locks called ordered sharing [AE90]. The resulting protocol, semantic locking with ordered sharing, provides more concurrency and hence minimizes blocking in object-oriented databases.

References

[AE90] D. Agrawal and A. El Abbadi. Locks with Constrained Sharing. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 85–93, April 1990. To appear in Journal of Computer and System Sciences.

[AE92] D. Agrawal and A. El Abbadi. A Non-restrictive Concurrency Control Protocol for Object Oriented Databases. In *Proceedings of the Third International Conference on Extending Data Base Technology (EDBT'92)*, volume 580 of *Lecture Notes in Computer Science*, pages 469–482. Springer-Verlag, March 1992. An extended version of this paper will appear in Distributed and Parallel Databases: An International Journal.

[BBG89] C. Beeri, P. A. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[BDK92] F. Bancilhon, C. Dolobel, and P. Kanellakis, editors. *Building an Object-Oriented database System: The story of O_2*. Morgan Kaufmann, 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[CF90] M. Cart and J. Ferrié. Integrating Concurrency Control into an Object-Oriented Database System. In *Proceedings of the International Conference on Extending Data Base Technology, Springer-Verlag, LNCS 416*, pages 363–376, 1990.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[FLW90] A. Fekete, N. Lynch, and W. Weihl. A Serialization Graph Construction for Nested Transactions. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 94–108, April 1990.

[GK88] J. F. Garza and W. Kim. Transaction Management in an Object-oriented Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–55, June 88.

[Gra78] J. N. Gray. Notes on database systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

[HDK+90] U. Herrmann, P. Dadam, K. Kuspert, E. Roman, and G. Schl ageter. A Lock Technique for Disjoint and Non-Disjoint Complex Objects. In *Proceedings of the International Conference on Extending Data Base Technology, Springer-Verlag, LNCS 416*, pages 219–237, March 1990.

[HH91] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43(1):2–24, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[HW91] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases*. The MIT Press, 1990.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.

[MRW+93] P. Muth, T. C. Rakow, G. Weikum, P. Brössler, and C. Hasse. Semantic Concurrency Control in Object-Oriented Database Systems. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 233–242, April 1993.

[Res94] R. Resende. *Conflict and Semantic Serializability of Nested transactions Protocols.* PhD thesis, Department of Computer Science, University of California at Santa Barbara, 1994. In preparation.