Context-specific synchronization for atomic data types in object-based databases

Man Hon Wong*, Divyakant Agrawal b,*

a Department of Computer Science, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
b Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

Abstract

Highly concurrent and reliable atomic data types are crucial for object-based databases. Deferred update (DU) and update-in-place (UIP) are two common recovery strategies for implementing atomic data types. These two strategies place incomparable constraints on the conflict relations between concurrent operations resulting in incomparable synchronization protocols. Also, the conflict relations used are usually static in the sense that they depend only on the operation types, and the algorithms do not use the context-specific information that may be available in the system. In this paper, a new synchronization mechanism that employs a hybrid scheme by using both DU and UIP is proposed. Furthermore, the protocol is dynamic in the sense that context-specific information is also used to determine conflict relations among concurrent operations. Another extension is the use of ordered shared relationships between locks to execute conflicting operations concurrently. The execution of operations is never delayed in the proposed protocol; however, the commitment of the transactions invoking these operations may be delayed due to the restriction imposed by the ordered shared relationships between locks. It is demonstrated that the sets of histories accepted by the two phase locking protocols using DU or UIP are proper subsets of the set of histories accepted by the proposed protocol.

1. Introduction

Atomic transactions are widely used for coping with concurrency and failures in database systems. Concurrency control and recovery are two of the main components of transaction management in databases. Concurrency control ensures the correct execution of a set of transactions, even when the operations of different transactions are interleaved. Recovery ensures the consistency of the database state, even when failures occur, or when transactions abort before completion. Traditionally, a database system is modeled as a collection of objects which can only be read or written by transactions [3, 4, 12]. More recently, a number of researchers have considered placing more structure on data objects and have shown how this structure can be used to permit more
The notion of atomic transactions is also used to implement atomic data types in distributed systems. In particular, the system is modeled as a collection of objects with abstract data-type specifications and synchronization algorithms for concurrency control and recovery exploit the semantics of data types.

A theory for analyzing the interrelationship between concurrency control and recovery protocols is developed by Weihl [17]. The author uses two-phase locking [6,9] for concurrency control and deferred update (DU) and update-in-place (UIP) [7,16] strategies for recovery. In DU, the effect of an operation is incorporated into an object when the transaction that executed the operation commits. Therefore, each operation is executed on a state which does not contain the effects of operations from other uncommitted transactions. In UIP, the effect of an operation is incorporated into an object immediately after the operation is executed. Therefore, an operation may be executed on a state which contains the effects of operations from other uncommitted transactions. Weihl shows that the weakest conflict relations for DU and UIP are the complement of the forward commutativity relation and the right backward commutativity relation, respectively. Therefore, the two recovery schemes place incomparable constraints on concurrency control resulting in incomparable synchronization protocols. Moreover, he showed that it is impossible to improve the concurrency by further weakening the conflict relations. However, the protocols presented in [17] are static in the sense that the algorithms do not use the context-specific information that may be available in the system. When context-specific information is used, we may improve the concurrency by weakening the conflict relation. Consider an example of a banking system with a locking protocol that uses UIP for recovery, where the granularity of locks is at object level and commutativity is used to derive conflicts among operations. In such a system, a successful withdraw operation conflicts with all concurrent deposit operations, since a withdraw operation in general does not commute with a deposit operation when they are executed with the update-in-place strategy (in which right backward commutativity is used). Consider the following execution of operations of three different transactions on an account with initial balance of $0:

- Transaction 1: deposit($10)
- Transaction 2: deposit($10)
- Transaction 3: withdraw ($8)

If we use the static notion of conflicts derived from the right backward commutativity, the withdraw operation conflicts with both deposit operations in the above execution. Note that if DU is used, the withdraw will be unsuccessful, and an unsuccessful withdraw operation also conflicts with both deposit operations. However, if the context of the operations is also taken into account, the withdraw operation may conflict only with the deposit operation of the first transaction but not with the deposit operation of the second transaction. In other words, if it is sure that the first deposit operation is serialized before the withdraw operation, there is no constraint on the serialization order of the second deposit operation with respect to the withdraw operation. In this paper, we develop a protocol that employs a dynamic notion of conflicts by using context-specific information for executing operations. The context-specific information
is derived from the views of the operations which can be determined dynamically in our protocol. The view of an operation is the state on which the operation is executed. In general, the context of an execution depends only on the history of the execution, it does not depend on the semantics of the transactions.

We use a hybrid scheme based on both deferred update and update-in-place. In the above example, the withdraw operation is executed with update-in-place strategy with respect to the first deposit operation while it is executed in deferred update strategy with respect to the second deposit operation. That is the withdraw operation is executed on a state which contains the effects of the first deposit operation, but does not contain the effects of the second deposit operation. From [17], it can be verified that indeed, the withdraw operation conflicts (since it does not right backward commute) with the first deposit operation while it does not conflict (since it commutes forward) with the second deposit operation.

Another extension that is used in this paper is that of using ordered shared relationships between locks in the two-phase locking protocol [1]. Note that if we employ the standard shared and exclusive relationships between locks to capture the no-conflict and conflict relations between operations, in the above example the withdraw operation cannot be executed concurrently with the two deposit operations since it conflicts with at least one deposit operation. Ordered shared relationships between locks permits concurrent execution of conflicting operations as long as the serialization order of the transactions is consistent with respect to the execution order of conflicting operations.

The paper is organized as follows. In the next section, we present the model of the system. In Section 3, we describe the specification and implementation of the proposed protocol. In Section 4, we present the proof of correctness of the proposed protocol and show that the proposed protocol permits more concurrency than both two-phase locking with deferred update and two-phase locking with updated-in-place. The recovery issues are discussed briefly in Section 5. We conclude with a discussion of our results in Section 6.

2. The database model

The model used in this paper is an extension of [17]. We assume that the database is a collection of objects derived from abstract data types. An abstract data-type determines a set of acceptable values and a set of allowed atomic operations for the objects. Users interact with the database by invoking transactions. A transaction can access an object or modify the state of the object by invoking operations defined for that object. The corresponding result of the invocation is returned to the transaction through a response message. Formally, an execution in the system consists of a set of events that are partially ordered. These events can be classified into the following types:

1. An invocation event, \( \langle \text{op}, A \rangle @X \), occurs when transaction \( A \) invokes an operation \( \text{op} \) on object \( X \).
2. A commit request event, \langle \text{ReqCommit}, A \rangle @X, occurs when transaction \( A \) requests its commitment at object \( X \).

3. A response event, \langle \text{res}, A \rangle @X, occurs when a response \( \text{res} \) to an earlier invocation or a commit request is returned from an object \( X \) to transaction \( A \).

4. A commit event, \langle \text{commit}, A \rangle @X, occurs when a commit message from transaction \( A \) is received.

5. An abort event, \langle \text{abort}, A \rangle @X, occurs when an abort message from transaction \( A \) is received.

We impose certain constraints on the execution of transactions:
- Each transaction \( A \) waits for the response to its last invocation before invoking the next operation or commit request.
- Each transaction \( A \) either commits on all objects or aborts on all objects.
- Each transaction \( A \) commits only after it has received responses to its commit requests from all the objects on which it performed operations.

Similarly, the following constraint is imposed on the objects:
- An object \( X \) can generate a response for transaction \( A \) only if \( A \) has a pending invocation at \( X \).

A history \( H \) is a sequence of events in an execution, where the order of the events is consistent with the partial order of the execution. A subhistory of \( H \) restricted to an object \( X \) (transaction \( A \)), denoted \( H|X \) (\( H|A \)), is defined as the subsequence of events that involve \( X \) (\( A \)) in \( H \).

A pair consisting of an invocation "\( \text{inv} \)" and the corresponding response "\( \text{res} \)" on an object \( X \) is termed as an operation, written as \( X: [\text{inv}, \text{res}] \). An operation sequence of a history is a series of operations produced by combining the corresponding invocations and responses with the order of the operations conforming to the order of the response events. A serial specification of an object \( X \), denoted \( \text{Spec}(X) \), defines the set of allowable operation sequences for that object. An operation sequence \( h \) of an object \( X \) is said to be valid if \( h \) belongs to \( \text{Spec}(X) \). In this paper, we assume that for any object \( X \), \( \text{Spec}(X) \) satisfies the prefix closure property, i.e. if an operation sequence \( h \in \text{Spec}(X) \), for any prefix \( g \) of \( h \), \( g \in \text{Spec}(X) \). A state of an object is simply an operation sequence, where the initial state is represented by the empty operation sequence. Of course, we can use a more efficient method for the implementation of an object’s state. The view of an operation is the state used to determine a response to the operation by executing the operation in that state. An operation \( p \) of object \( X \) is said to be valid on a view \( v \), if \( v \cdot p \in \text{Spec}(X) \). For example, in DU the view of an operation contains only committed operations; while in UIP the view of an operation contains all the un aborted operations. As will become apparent later, the view of an operation in the proposed protocols can be constructed dynamically.

Two operation sequences \( h_1 \) and \( h_2 \) are said to be equivalent (denoted by \( h_1 \equiv h_2 \)) if, for any operation sequence \( g \), \( h_1 \cdot g \) is valid if and only if \( h_2 \cdot g \) is valid. Note that a dot "\( \cdot \)" means concatenation of operation sequences. Intuitively, two operation sequences are equivalent if they lead the object to the states that are indistinguishable from the responses of further operations executed on the object. We say that two operations \( p \)
and $q$ commute forward with each other if for any operation sequence $h$ such that $h \cdot p$ and $h \cdot q$ are valid, then $h \cdot p \cdot q$ and $h \cdot q \cdot p$ are valid and equivalent. An operation $q$ is said to right commute backward with another operation $p$ if, for any operation sequence $h$ such that $h \cdot p \cdot q$ is valid, then $h \cdot q \cdot p$ is valid and equivalent to $h \cdot p \cdot q$. Note that forward commutativity is a symmetric relation while right backward commutativity is an asymmetric relation.

In this paper, bank accounts are used as the data objects in the examples. The state of a bank account can be represented by the amount of money in the account, since all the operation sequences which bring the money in the account to the same amount are equivalent. At any time, the amount of the money cannot be negative. Three types of operations, namely, deposit, withdraw, and balance can be invoked by transactions. The response of a deposit operation is always ok. The response of a withdraw operation may be ok or no whereas the response of a balance operation is the amount of the money in the account. The forward commutativity relation and the right backward commutativity relation for the operations of a bank account are depicted in Tables 1 and 2, respectively. In Table 1, a “yes” entry indicates that the operations for the given row and column commute forward with each other and an empty entry indicates that the operations do not commute forward. For instance, operation $BA:[withdraw(j), ok]$ commutes forward with operation $BA:[deposit(i), ok]$, because for any state $s$, if both $s \cdot BA:[withdraw(j), ok]$ and $s \cdot BA:[deposit(i), ok]$ are valid, $s \cdot BA:[withdraw(j), ok] \cdot BA:[deposit(i), ok]$ and $s \cdot BA:[deposit(i), ok] \cdot BA:[withdraw(j), ok]$ are valid and equivalent.

In Table 2, on the other hand, a “yes” entry indicates that the operation on the row of the entry right commutes backward with the operation on the column of the entry and an empty entry indicates that the operation on the row of the entry does not right commute backward with operation on the column of the entry. For example, operation $BA:[withdraw(j), ok]$ does not right commute backward with operation $BA:[deposit(i), ok]$, since the condition that $s \cdot BA:[deposit(i), ok] \cdot BA:[withdraw(j), ok]$ is valid does not in general imply $s \cdot BA:[withdraw(j), ok] \cdot BA:[deposit(i), ok]$ is valid. In particular, the operation sequence $BA:[deposit(100), ok] \cdot RA:[withdraw(100), ok]$ is valid on a bank account with $0$, but the operation sequence $BA:[withdraw(100), ok] \cdot BA:[deposit(100), ok]$ is not valid on the same account with the same state.
3. The protocol

We first give an overview to motivate the approach that will be used in the protocol. Next, we provide the abstract specifications of the protocol and present an implementation using ordered shared relationships between locks. We conclude this section with an example to illustrate the protocol.

3.1. Overview

Update-in-place (UIP) and deferred update (DU) are two common implementations for recovery protocols in database systems. In UIP, a current state is maintained for each data object. When an operation is executed, the response to the operation is determined on the basis of the current state and the state is updated immediately after the execution of the operations. In DU, transactions can be perceived as executing operations in their private workspaces. The response of an operation is determined by the committed operations and all the previous operations of the same transaction. It has been shown that an implementation of UIP is correct if the conflict relation used for concurrency control includes the complement of the right backward commutativity relation of the operations whereas an implementation of DU is correct if the conflict relation includes the complement of the forward commutativity relation of the operations [17].

Let \( h \) be the operation sequence involving operations of committed transactions on an object and let \( C(p) \) be the set of uncommitted operations (concurrent set) when an operation \( p \) of transaction \( A \) is executed on that object. For brevity, assume that \( p \) is the only operation executed on the object by \( A \). In UIP, the view of \( p \) is \( h \) as well as all the operations in \( C(p) \), and \( p \) must right commute backward with all the operations in \( C(p) \). On the other hand, in DU, the view of \( p \) is \( h \), and \( p \) must commute forward with all the operations in \( C(p) \).

We propose a protocol that uses the combined notion of UIP and DU. The view of operation \( p \) includes \( h \) and a set of uncommitted operations \( C_r(p) \) such that \( C_r(p) \subseteq C(p) \). \( C_r(p) \) is the set of uncommitted operations which belongs to the view of \( p \). Actually, \( C_r(p) \) is an ordered set (sequence), but we are ignoring this issue for the time being. The criterion for selecting \( C_r(p) \) is that \( p \) must right commute backward with all the operations in \( C_r(p) \) and must commute forward with all the
operations in $C(p) \setminus C_v(p)$. If such a $C_v(p)$ cannot be found, the execution of $p$ will be delayed. Consider, for example the following history:

\[
\langle \text{deposit}(9), A \rangle @ BA \\
\langle \text{ok}, A \rangle @ BA \\
\langle \text{withdraw}(8), B \rangle @ BA \\
\langle \text{ok}, B \rangle @ BA \\
\langle \text{withdraw}(20), C \rangle @ BA \\
\langle \text{no}, C \rangle @ BA \\
\langle \text{CommitReq}, A \rangle @ BA \\
\langle \text{ok}, A \rangle @ BA \\
\langle \text{commit}, A \rangle @ BA \\
\langle \text{CommitReq}, B \rangle @ BA \\
\langle \text{ok}, B \rangle @ BA \\
\langle \text{commit}, B \rangle @ BA \\
\langle \text{CommitReq}, C \rangle @ BA \\
\langle \text{ok}, C \rangle @ BA \\
\langle \text{commit}, C \rangle @ BA
\]

The corresponding operation sequence of the above history is

\[
op_A BA: [\text{deposit}(9), \text{ok}] \\
op_B BA: [\text{withdraw}(8), \text{ok}] \\
op_C BA: [\text{withdraw}(20), \text{no}]
\]

Assume that the initial amount in $BA$ is $10$. Since operation $BA: [\text{withdraw}(8), \text{ok}]$ does not right commute backward with operation $BA: [\text{deposit}(9), \text{ok}]$, the above history cannot be accepted by a protocol that uses UIP. Similarly, $BA: [\text{withdraw}(20), \text{no}]$ does not commute forward with $BA: [\text{deposit}(9), \text{ok}]$, hence, the history cannot be accepted by a protocol that employs DU.

If the context in which operations are executed is considered, the above operations commute with each other. In particular, when $op_A = BA: [\text{deposit}(9), \text{ok}]$ is executed, its view is the empty operation sequence, denoted by $A$, with the initial value of $BA$ being $10$. The operation $op_A$ is valid on $A$, and both $C_v(op_A)$ and $C(op_A)$ are empty. Next, $op_B = BA : [\text{withdraw}(8), \text{ok}]$ is valid on $A$ and it commutes forward with $op_A$. Thus, $C_v(op_B) = \emptyset$ and $C(op_B) = \{op_A\}$. Finally, the last operation $op_C = BA : [\text{withdraw}(20), \text{no}]$ commutes forward with $op_B$ but does not commute forward with $op_A$. Thus, the view of $op_C$ required by the protocol is $A$ followed by $op_A$. It can be verified that $op_C$ is valid on this view and, hence, $C_v(op_C) = \{op_A\}$ and $C(op_C) = \{op_A, op_B\}$. Therefore, the above history can be accepted by using the combined notion of UIP and DU.

The above example illustrates that to execute an operation $p$, $C(p)$ be partitioned such that $p$ right backward commutes with all the operations in $C_v(p)$ which do not belong to the same transaction of $p$ and all the operations in $C(p) \setminus C_v(p)$ not belonging to the same transaction of $p$ commute forward with $p$. To sum up, we have proposed a generalized method to determine whether an operation conflicts with
some of the operations in the concurrent set (the set of active operations). On one extreme, if a protocol always enforces that $C_r(p) = C(p)$, i.e., the view of $p$ contains all the concurrent operations, the protocol reduces to the one using the update-in-place recovery strategy. On the other extreme, if the protocol requires that $C_r(p)$ contains only those operations in the same transaction of $p$, the protocol reduces to the one using the deferred update recovery strategy.

Next, we present a protocol which further relaxes the criterion for selecting the view of an operation. The protocol permits the execution of $p$ even when $p$ does not right commute backward with all the operations in $C_r(p)$, i.e., $p$ conflicts with some of the operations in the concurrent set. This is accomplished by introducing a new relationship between locks called ordered sharing [1].

3.2. Specification and implementation of the protocol

We use a newly introduced relation called ordered shared [1] relation for executing conflicting operations concurrently. In particular, if $q$ conflicts with $p$ and the lock of $q$ is acquired after the lock of $p$ has been acquired, then we say that $q$ has an ordered shared relation with respect to $p$ denoted by $p \rightarrow q$. The restrictions imposed by the ordered shared relations are that the execution of $q$ must follow the execution of $p$, $q$ must include $p$ in constructing its view, and the transaction invoking $q$ can commit only after the transaction invoking $p$ has terminated. Depending on the semantics of the operations $p$ and $q$, the transaction invoking $q$ may have to be aborted if the invoker of $p$ aborts. An operation $q$ has shared relations with all the operations that do not conflict with $q$ and has ordered shared relations with all the operations that conflict with $q$.

Let $C(p)$ and $C_r(p)$ be defined as in the previous section; i.e., $C(p)$ is the concurrent set of the object when $p$ is executed and $C_r(p)$ is a subset of $C_r(p)$ such that all the operations in $C_r(p)$ belongs to the view of $p$ and all the operations in $C(p) \setminus C_r(p)$ do not belong to the view of $p$. Furthermore, let $<_{C(p)}$ be the partial order determined by the ordered shared relations among the operations in $C(p)$ and $OpSeq_r(p)$ be the operation sequence involving operations in $C_r(p)$ that is consistent with $<_{C(p)}$. The specifications of assigning the relations between a new operation $p$ invoked by a transaction $A$ and other operations executed on the object $X$ are as follows:

1. $C_r(p)$ contains at least all the previous operations invoked by $A$ and all the operations in $C(p)$ that do not commute forward with $p$. If an operation $s$ is contained in $C_r(p)$ and there exists an operation $r \in C(p)$ such that $r \rightarrow s \in <_{C(p)}$, then $r$ must be contained in $C_r(p)$ and appears before $s$ in $OpSeq_r(p)$.

2. $p$ must be valid on its view, i.e., if $h$ is the sequence of the committed operations, then $h \cdot OpSeq_r(p) \cdot p \in Spec(X)$.

3. $p$ has ordered shared relations with respect to all the committed operations, all the previous operations invoked by $A$, and all the operations in $C_r(p)$ with which $p$ does not right commute backward. (All the other operations may have shared relations with $p$.)
Note that in the specification, the view of an operation is chosen according to the commutativity property of the operation by the scheduler when a new invocation is received. Therefore, the response of the new operation has to be known before the view is chosen, since an operation contains both the invocation and the response. However, the response is determined by the view. This leads to the problem of circularity. There are several implementations that can be used to circumvent this problem. One of the simple techniques is to determine the response by including all the uncommitted operations in the view. After the response is known, we can refine the view such that it includes less operations and satisfies the above specifications. For example, after the response is known we can remove those concurrent operations which commute forward with \( p \) and with which \( p \) does not right commute from the view whenever \( p \) is still valid on the resulting view.

The protocol can be implemented by a strict two-phase locking protocol that uses two types of lock relationships: shared and ordered shared relationships between locks [1]. When an operation is executed on an object, it obtains locks in shared relationship with respect to all the operations with which it has shared relations and obtains locks in ordered shared relationship with respect to all the concurrent operations with which it has ordered shared relations.

We present our protocol by describing how various operations are executed on an object. Let \( \text{pending}(A) \) be the pending invocation (which has not been responded) of transaction \( A \) at object \( X \); \( \text{lock}(p, q) \) be the lock relationship of an operation \( p \) with respect to another operation \( q \), \( \text{opseq}(A) \) be the sequence of operations executed by a transaction \( A \) on object \( X \); and \( \text{ord}(h, C, p) \) be the set of operations on which \( p \) has ordered shared relations when \( p \) is executed on the object according to the above specifications where \( h \) and \( C \) are the committed operation sequence and the concurrent set in \( X \), respectively. The protocol is presented in Fig. 1 where it is assumed that the code associated with each event is executed atomically.

An invocation event occurs when an object receives an invocation from a transaction \( A \). The invocation that has not been responded is referred to as the pending invocation of transaction \( A \). Once an invocation is received, the code associated with the response event is triggered by the scheduler to execute the invocation on the state that includes the effects of all the committed and concurrent operations; i.e., \( h \cdot C_L \). The invocation together with the response form the new operation \( p \). The locking relationships between operation \( p \) and other concurrent operations are assigned according to the specification as described above. Finally, the response will be returned to transaction \( A \), i.e. the response event occurs. A commit request event occurs when a commit request is received from a transaction \( A \). If none of the operations from transaction \( A \) has an ordered shared relationship with respect to an active operation of another transaction, a positive response will be returned to transaction \( A \), hence the response event occurs. A commit event occurs when a commit message is received from a transaction. The effects of the operations from the transaction will be incorporated to the committed state \( (h) \) of the object. And the operations will be removed from the concurrent set. Similarly, an abort event occurs when an abort message is received from a transaction.
\langle op, A \rangle @X:
pending(A) \leftarrow op

\langle ReqCommit, A \rangle @X:
pending(A) \leftarrow ReqCommit

\langle res, A \rangle @X:
IF pending(A) \neq ReqCommit THEN
    let \( C_L \) = a sequence containing all the operations in \( C \) with the order consistent with the ordered shared relations
    let \( p = (pending(A), res) \) such that \( h \cdot C_L \cdot p \in Spec(X) \)
    FOR all \( q \in C \) DO
        IF \( q \in ord(h, C, p) \) THEN
            lock\((p, q)\) = ordered-shared
        ELSE
            lock\((p, q)\) = shared
        END (* IF *)
    END (* FOR *)
    \( C \leftarrow C \cup p \)
    send \( res \) to \( A \)
ELSE IF pending(A) = ReqCommit THEN
    IF \( \{q | q \in C, p \in opseq(A), lock(p, q) = ordered-shared\} = \emptyset \) THEN
        send ok to \( A \)
    END (* IF *)
END (* IF *)

\langle Commit, A \rangle @X:
\( h \leftarrow h \cdot opseq(A) \)
\( C \leftarrow C \setminus opseq(A) \)

\langle Abort, A \rangle @X
\( C \leftarrow C \setminus opseq(A) \)

---

Fig. 1. Execution of operations of transaction \( A \) at object \( X \)

The effects of the operations from the transaction will be ignored and the operations will be removed from the concurrent set.

Recall that a transaction can commit only after it receives response for commit requests from all the objects accessed by the transaction. The transaction is blocked if any of the objects delays such a response because the transaction acquired a lock in ordered shared relationship with respect to another transaction and the latter transaction has not yet terminated, i.e., committed or aborted. In this case, the former transaction
is said to be on hold. Therefore, the commit order of transactions on an object is consistent with the order of the ordered shared relations on that object. In addition, if ordered shared locks are not used, a new operation $p$ has to be delayed when $ord(h, C, p) \neq \emptyset$, i.e., $p$ conflicts with some of the concurrent operations.

3.3. An example

We use an example to illustrate the execution of transactions in the proposed protocol. Suppose we have the following operation sequence on a bank account where each operation is executed by a different transaction:

1. $BA : [\text{deposit}(10), \text{ok}]$
2. $BA : [\text{deposit}(9), \text{ok}]$
3. $BA : [\text{withdraw}(8), \text{ok}]$
4. $BA : [\text{withdraw}(9), \text{ok}]$
5. $BA : [\text{withdraw}(4), \text{no}]$

Assume that the initial amount of money in the account is $0$, and all the operations are executed concurrently by different transactions. Lock diagrams are used to show the lock relations of the intermediate states. $p \rightarrow q$ means the operation $q$ has an ordered shared relation with respect to $p$. If there is no edge between two operations, they are assumed to have a shared relation. All the uncommitted operations, by default, have ordered shared relation with respect to the committed operations. For brevity, these relations are not shown in the diagrams. However, we do put arrows from the initial state to the operations which take the initial state directly as their views. The initial state is represented by a small circle.

$BA : [\text{deposit}(10), \text{ok}]$ is valid on the initial state, and no other concurrent operation exists. Thus, it can be executed and its lock has ordered shared relations with all the committed operations.

$BA : [\text{deposit}(9), \text{ok}]$ commutes forward with $BA : [\text{deposit}(10), \text{ok}]$. So, it can choose the initial state as its view. Obviously, it is valid on the initial state. Therefore its lock has ordered shared relations with the committed operations, and has a shared relation with the other deposit operation. The lock relations up to this operation are shown in Fig. 2.

![Fig. 2. Lock relation after operations 1 and 2.](image)

The successful withdraw operation $BA : [\text{withdraw}(8), \text{ok}]$ commutes forward with a deposit operation. So, it is not required to include the deposit operations in its view. However, this operation is invalid on the initial state. It becomes valid after a deposit operation, say $BA : [\text{deposit}(10), \text{ok}]$, is included in its view. Since a successful withdraw operation does not right commute backward with a deposit operation, $BA : [\text{withdraw}(8), \text{ok}]$ has an ordered shared relation on $BA : [\text{deposit}(10), \text{ok}]$. The
lock relations are shown in Fig. 3. Note that the withdraw operation depends on only
one deposit operation and is independent of the other deposit operation because the
other deposit operation is not included in the view of the withdraw operation and it
commutes forward with the withdraw operation. Moreover, if the initial amount of
money in the account is greater than or equal to $8, the withdraw operation can even
have shared relations with both deposit operations. This illustrates that conflict relations
depend not only on the operation types but also on the context of the operations.

\[
\text{[deposit(10), ok] } \rightarrow \text{ [withdraw(8), ok]}
\]

\[
\text{[deposit(9), ok]}
\]

Fig. 3. Lock relation after operations 1, 2 and 3.

Since the successful withdraw operation \( BA : [\text{withdraw}(9), \text{ok}] \) does not commute
forward with another successful withdraw operation, \( BA : [\text{withdraw}(8), \text{ok}] \) has to be
included in the view of \( BA : [\text{withdraw}(9), \text{ok}] \). Now, the view contains the initial
state followed by \( BA : [\text{deposit}(10), \text{ok}] \) and \( BA : [\text{withdraw}(8), \text{ok}] \). However, \( BA : [\text{withdraw}(9), \text{ok}] \)
is invalid on the view. The operation becomes valid when \( BA : [\text{deposit}(9), \text{ok}] \) is also included in its view. A successful withdraw operation right
commutes backward with another successful withdraw operation, but does not right
commute backward with a deposit operation. Therefore, \( BA : [\text{withdraw}(9), \text{ok}] \) has a
shared relation with \( BA : [\text{withdraw}(8), \text{ok}] \), but has ordered shared relations with the
two deposit operations. The lock relations are shown in Fig. 4. Note that although
\( BA : [\text{withdraw}(8), \text{ok}] \) is required to be contained in the view of \( BA : [\text{withdraw}(9), \text{ok}] \)
because the two withdraw operations do not forward commute, yet it has a shared rela-
tion with \( BA : [\text{withdraw}(9), \text{ok}] \) because a withdraw operation right backward commutes with another withdraw operation.

\[
\text{[deposit(10), ok] } \rightarrow \text{ [withdraw(8), ok]}
\]

\[
\text{[deposit(9), ok] } \rightarrow \text{ [withdraw(9), ok]}
\]

Fig. 4. Lock relation after operations 1, 2, 3 and 4.

An unsuccessful withdraw operation does not commute forward with a deposit op-
eration. Thus, the view of \( BA : [\text{withdraw}(4), \text{no}] \) consists of the initial state followed
by both the deposit operations. However, \( BA : [\text{withdraw}(4), \text{no}] \) is invalid on the view.
It becomes valid after the two withdraw operations are included in its view. Since
an unsuccessful withdraw operation right commutes backward with a deposit oper-
aration, but does not right commute backward with a successful withdraw operation,
\( BA : [\text{withdraw}(4), \text{no}] \) has shared relations with the deposit operations and has ordered
shared relations with the two successful withdraw operations. The lock relations are
shown in Fig. 5. Although \( BA : [\text{withdraw}(4), \text{no}] \) has no direct ordered shared relation
with respect to the two deposit operations, it has ordered shared relations with them
transitively. The above example illustrates that the execution of operations is never
delayed in the proposed protocol, however, the commitment of the transactions invoking these operations may be delayed due to the ordered shared relations. Since the ordered shared relations are minimized by exploiting the context-specific information among operations, the commit delays will be minimal. In particular, if a transaction is delayed from committing in our protocol then that transaction would have been blocked from executing its operations in other protocols [17]. The above execution cannot be accepted by Weihl’s protocols, because when UIP is used, successful withdraw operations conflict (do not right backward commute) with deposit operations and when DU is used, two successful withdraw operations are conflicting (do not commute forward).

4. Proof of correctness

In this section, we demonstrate the correctness of the proposed protocol. We first show that an operation sequence projected on an object is valid if the order is consistent with the order of the ordered shared relations on the object. Next, we prove that the commitment order of transactions is consistent with the order of the ordered shared relations on each object. Since the transactions are serialized in the commitment order at every object, the resulting execution is serializable. We also demonstrate that all the histories accepted by the two-phase locking protocol using either DU or UIP are also accepted by the proposed protocol even without using ordered shared locks.

Definition 1. A dependency order $V$ is a partial order on a set of operations executed on an object. The ordered pair $(q, p)$ belongs to the dependency order if and only if operation $p$ has an ordered shared relation with operation $q$, i.e. $q \rightarrow p$.

Definition 2. An operation sequence $h$ is said to be consistent with a dependency order $V$, if an operation $p$ is in $h$ then for all $(q, p) \in V$, $q$ is also in $h$ and appears before $p$ in $h$.

Proposition 1. Prefix Closure Property. Any prefix of a valid operation sequence is also valid.

Theorem 1. If $h \cdot p$ and $h \cdot q_1q_2\ldots q_n$ are valid, and $p$ commutes forward with $q_1, q_2, \ldots, q_n$, then $h \cdot q_1q_2\ldots q_np$ is valid.

Proof. We prove the theorem by induction. If $n = 1$, from the definition of forward commutativity, the theorem follows. Assume that the theorem holds for $n = k$. Suppose $h \cdot p$ and $h \cdot q_1q_2\ldots q_{k-1}$ are valid. From the prefix closure property, $h \cdot
$q_1q_2\ldots q_k$ is also valid. Hence, from the induction hypothesis, $h \cdot q_1q_2\ldots q_k p$ is valid. Let $h' = h \cdot q_1q_2\ldots q_k$, therefore $h' \cdot p$ and $h' \cdot q_{k-1}$ are valid. From the definition of forward commutativity property, $h' \cdot q_{k+1} \cdot p$ is valid; i.e. $h \cdot q_1q_2\ldots q_{k-1} p$ is valid. ☐

**Theorem 2.** *Any operation sequence consistent with a dependency order is valid. Moreover, any two operation sequences containing the same set of operations and consistent with the dependency order are equivalent.*

**Proof.** Let $V_n$ be the dependency order of an object after a set of $n$ operations, $O_n$, has been executed on the object. Let $\mathcal{H}_n$ be the set of operation sequences consistent with the dependency order $V_n$. We prove the theorem by induction.

The basis case when $n = 1$ holds since all the operation sequences in $\mathcal{H}_1$ involve a single operation and hence are valid. For the induction hypothesis, assume that the theorem holds for $n = k$, i.e., all the sequences in $\mathcal{H}_k$ are valid and any two sequences in $\mathcal{H}_k$ containing the same set of operations are equivalent.

Suppose now, there is a new operation $p$ executed on the object. This operation may have ordered shared relations with other operations. Therefore,

$$V_{k+1} = V_k + \{(q, p) \mid q \in O_k \land q \rightarrow p\}$$

For each operation sequence $h \in \mathcal{H}_{k+1}$, we have the following cases:

- **Case I:** All operations in $h$ belong to $O_k$.
- **Case II:** Operation sequence $h$ is of the form $h' \cdot p$ and all operations in $h'$ are in the view of $p$.
- **Case III:** Operation sequence $h$ is of the form $h' \cdot p$ and $h'$ may contain operations not in the view of $p$.
- **Case IV:** Operation sequence $h$ is of the form $h' \cdot p \cdot h''$.

We first prove that the operation sequences in the above four cases are valid. Then, we prove that any two operation sequences involving the same set of operations in $\mathcal{H}_{k+1}$ are equivalent.

**Proof of Case I.** All operations in $h$ belong to $O_k$. Therefore, $h \in \mathcal{H}_k$ and hence from the induction hypothesis $h$ is valid.

**Proof of Case II.** $h = h' \cdot p$. Let $q_1h_1q_2h_2\ldots q_mh_m$ be the view of $p$, where $h_1h_2\ldots h_m$ contains the same set of operations as $h'$. Note, $q_i$ has shared relations with $p$ and all operations in $h_{i+1}h_{i+2}\ldots h_m$. If $q_i$ does not have a shared relation with any of the above operations $o$, then either $(q_i, o)$ or $(o, q_i)$ is in the dependency order $V_{k-1}$. However, if $(q_i, o) \in V_{k+1}$ then $h'$ will not be consistent with the dependency order. On the other hand, if $(o, q_i) \in V_{k+1}$ then $q_1h_1q_2h_2\ldots q_mh_m$ cannot be the view of $p$. Thus, $h' \cdot q_1q_2\ldots q_m$ is also consistent with the dependency order, $V_k$ (since $p$ is not contained in the sequence), hence $h' \cdot q_1q_2\ldots q_m \equiv q_1h_1q_2h_2\ldots q_mh_m$. Since $p$ is valid
on its view, \( h' \cdot q_1 q_2 \ldots q_m p \) is valid. Moreover, from the specification of the protocol, \( p \) right commutes backward with \( q_1 \), therefore, \( h' \cdot p q_1 q_2 \ldots q_m \) is valid. By the prefix closure property, \( h' \cdot p \) is also valid.

**Proof of Case III.** \( h = h' \cdot p \) and \( h' \) may contain some operations not in the view of \( p \). Note that all operations not contained in the view of \( p \) must have shared relations with \( p \). Moreover, they commute forward with \( p \).

Let \( h' = h_1 q_1 h_2 q_2 \ldots h_m q_m h_{m+1} \), where \( h_i, i = 1 \ldots m + 1 \) contain operations in the view of \( p \) and \( q_i, i = 1 \ldots m \) are the operations not contained in the view of \( p \). Since \( q_i \) does not belong to the view of \( p \), for any operation \( o \) in \( h_1 \ldots h_{m+1}, (q_i, o) \notin V_k \). Thus, \( h_1 \ldots h_{m+1} q_1 \ldots q_m \) is consistent with the dependency order \( V_k \), hence, \( h' \equiv h_1 \ldots h_{m+1} q_1 \ldots q_m \). From the result of case II, \( h_1 \ldots h_{m+1} \cdot p \) is valid. Moreover, from the specification of the protocol \( q_i \) commutes forward with \( p \). Therefore, from Theorem 1, \( h_1 h_2 \ldots h_{m+1} \cdot q_1 q_2 \ldots q_m p \) is valid, hence \( h' \cdot p \) is valid.

**Proof of Case IV.** \( h = h' \cdot p \cdot h'' \). Since \( h \) is consistent with the dependency order, \( p \) has shared relations with all the operations in \( h'' \). Let \( h'' = q_1 q_2 \ldots q_m \). From the result of case III, we have \( h' \cdot p \) and all operation sequences \( h' \cdot q_1 q_2 \ldots q_ip, \) where \( 1 \leq i \leq m, \) are valid. If \( p \) right commutes backward with \( q_m \), we have \( h'q_1 \ldots q_{m-1} pq_m \) and both of them are valid. Now, consider the case when \( q_m \) commutes forward with \( p \). Since \( hq_1 \ldots q_{m-1}q_m \in H_k \), it is valid. In addition, from above we know that \( hq_1 \ldots q_{m-1}p \) is valid. Hence by the forward commutativity property, we have \( h \cdot q_1 \ldots q_{m-1}p q_m = h \cdot q_1 \ldots q_{m-1}q_m p \). In other words, no matter if the shared relation is obtained from the forward commutativity property or from the right backward commutativity property, we can swap \( p \) with \( q_m \) and obtain an equivalent sequence. Similarly, we can swap \( p \) with \( q_{m-1} \) in the next step and so on. If the above step of swapping operations is repeated, eventually, we will have

\[
\begin{align*}
\begin{array}{c}
h' \cdot p \cdot h'' \\
\equiv h' \cdot h'' \cdot p
\end{array}
\end{align*}
\]

Therefore, \( h' \cdot p \cdot h'' \) is valid.

Since the operation sequences in each of the four cases are valid and the union of the operation sequences in the four cases is equal to \( H_{k+1} \), we can conclude that all the operation sequences in \( H_{k+1} \) are valid.

Now, we prove for any two operation sequences in \( H_{k+1} \) containing the same set of operations are equivalent. If the two operation sequences do not contain the operation \( p \), they are in \( H_k \), hence from the induction hypothesis they are equivalent. Consider the case that the operation \( p \) is contained in the operation sequences. Let the two operation sequences be \( h' \cdot p \cdot h'' \) and \( g' \cdot p \cdot g'' \). From the result of case IV, we have \( h' \cdot p \cdot h'' \equiv h' \cdot h'' \cdot p \) and \( g' \cdot p \cdot g'' \equiv g' \cdot g'' \cdot p \). From the induction hypothesis, \( h' \cdot h'' \equiv g' \cdot g'' \), therefore, \( h' \cdot h'' \cdot p \equiv g' \cdot g'' \cdot p \). Hence \( h' \cdot p \cdot h'' \equiv g' \cdot p \cdot g'' \). \( \square \)
Suppose $H$ is a history, and $L$ is a total order defined over the transactions in $H$. $H$ is said to be serializable in the order $L$ if the operation sequence constructed by reordering $H$'s operations in the induced order of $L$ is valid.

**Definition 3.** $\text{ser}(H,L)$ is a sequence of operations constructed by reordering and combining the invocation and response pairs in history $H$ such that its order agrees with the order when the transactions are executed serially in the order of $L$.

**Theorem 3.** A history $H$ is serializable, if there exists a total order $L$ over the transactions in $H$ such that for any object $X$, $\text{ser}(H|X,L)$ is valid and equivalent to the operation sequence corresponding to $H|X$.

This theorem appears as Lemma 3.2 in [15].

**Definition 4.** A lock relinquishing order is a partial order defined over a set of transactions such that a transaction $T_i$ is before another transaction $T_j$ if there exists an object accessed by both $T_i$ and $T_j$ and $T_i$ relinquishes its first lock before $T_j$ relinquishes any of its locks.

Note, since strict two-phase locking scheme is used, i.e., all locks are released when the transaction terminates, the lock relinquishing order is the same as the commitment order.

**Theorem 4.** If $H$ is a history and $L$ is a total order consistent with the lock relinquishing order of an object $X$, then $\text{ser}(H|X,L)$ is consistent with the dependency order of the object $X$.

**Proof.** Suppose $p_i$ and $p_j$ are operations in $\text{ser}(H|X,L)$ of transactions $T_i$ and $T_j$, respectively. If $T_i$ is before $T_j$ in $L$, then there must not be the case that $p_j \rightarrow p_i$; otherwise, $T_i$ cannot relinquish its first lock before $T_j$ relinquishes any of its locks.

Therefore, if $p_j \rightarrow p_i$ and $p_i$ is in $\text{ser}(H|X,L)$ (i.e. $T_i$ is in $L$), then $T_j$ must exist and is before $T_i$ in $L$. Hence, $p_j$ exists and is before $p_i$ in $\text{ser}(H|X,L)$. Thus, $\text{ser}(H|X,L)$ is consistent with the dependency order.

**Theorem 5.** A history $H$ accepted by the protocol can be serialized in a total order $L$ consistent with the lock relinquishing order.

**Proof.** From Theorem 4, for any object $X$, $\text{ser}(H|X,L)$ is consistent with the dependency order of $X$. Hence, from Theorem 2 for any objects $X$, $\text{ser}(H|X,L)$ is valid and equivalent to the operation sequence corresponding to $H|X$. Therefore, by Theorem 3, $H$ is serializable in the total order $L$.

The next two theorems demonstrate that the proposed protocol is at least as permissive as two phase locking with either DU or UIP.
Theorem 6. All the histories accepted by two-phase locking with DU are accepted by the proposed protocol even without using ordered shared locks.

Proof. We prove the theorem by induction on the length of the history. When the length of a history is zero (empty history), the history is accepted by both DU and our protocol. In addition, no ordered shared relation exists. Assume that the theorem is true for the histories with length = k and no ordered shared relation exists between operations of two different concurrent transactions. Suppose, there is a history accepted by DU and with length = k + 1. Let h be the operation sequence for the committed operations and \( C(p) \) be the concurrent set right before the last event occurs. By the induction hypothesis, the history without the last event is also accepted by our protocol and no ordered shared relation exists between operations of two different concurrent transactions. Moreover, the committed operation sequence, the concurrent set and the pending invocations are exactly the same as in the case of DU, because all these values depend on the history but not on the protocols. Now, consider the last event. According to the type of the last event, we have the following three cases:

- Invocation event: Any invocation event can be accepted by our protocol.
- Commit or abort event: Since no ordered shared relation exists between operations of two different concurrent transactions, any commit or abort event accepted by DU is also accepted by our protocol.
- Response event: If the response is accepted by DU, the corresponding operation, \( p \), commutes forward with all the operations in \( C(p) \), and is valid with \( h \) as its view. Therefore, in our protocol \( C_r(p) \) is empty or contains the operations in the same transaction of \( p \) only. Thus, \( p \) has shared relations with all the other concurrent operations. Hence, the response is accepted by our protocol. Moreover, no ordered shared relation between operations of two different concurrent transactions is issued.

Hence, if a history is accepted by two-phase locking with DU, it will be accepted by the proposed protocol.

Theorem 7. All the histories accepted by UIP are accepted by the proposed protocol even without using ordered shared lock mode.

Proof. The proof is exactly the same as the case of DU except in the case of the response events. If the response is accepted by UIP, the corresponding operation, \( p \), right commutes backward with all the operations in \( C(p) \), and \( p \) is valid on the view containing \( h \) followed by a permutation of the operations in \( C(p) \). By the right backward commutativity property, \( p \) is also valid on the view containing \( h \) followed by any combination of the operations in \( C(p) \). Thus, no matter how \( C_r(p) \) is chosen, \( p \) must be valid on the corresponding view. Since \( p \) right commutes backward with all the operations in \( C \), it right commutes backward with all the operations in \( C_r(p) \) (note that \( C_r(p) \subseteq C \)). Therefore, the response is accepted by the protocol and no ordered shared relation between operations of two different concurrent transactions exists. Hence, if a
history is accepted by two-phase locking with UIP, it will be accepted by the proposed protocol.

By using the results from [1], it is easy to show that context-specific two-phase locking protocol with ordered shared lock relations is more permissive than that with shared and exclusive lock relations.

5. Recovery

In this paper, a hybrid scheme of DU and UIP is proposed to increase the concurrency. The idea of the protocol is to generalize the approaches for constructing the view of an operation. Usually, if the view of a new operation contains all the other active operations and the state of the object is updated immediately (UIP), the current state which includes the effect of the active operations is stored in a stable storage and undo log is used for recovery. After a crash, the effect of all the aborted operations will be undone. On the other hand, if the view of a new operation contains the committed state followed by all the previous operations executed by the same transaction, and the effect of the operations are incorporated into the state of the object at commitment (DU), the committed state is stored in a stable storage and intentions list is used to cope with recovery. No recovery action is needed to recover the state of the object after a crash.

The reason for storing the current state in a stable storage for UIP and storing the committed state in a stable storage for DU is obvious. It is because the view of a new operation can easily be constructed from the state stored in the stable storage. Indeed, the recovery scheme can be chosen independent of the way in which the views of operations are constructed, however the recovery methods may have implications on the performance of the system. For example, when UIP is used to construct the view of an operation, intentions list can be used for recovery in which the committed state is stored in a stable storage and the current state is maintained in the volatile storage for constructing the view of a new operation. Similarly, when DU is used to construct the view of an operation, we can store the current state in a stable storage and use undo log for recovery.

In our protocol, the view of a new operation contains the committed state and a subset of active operations. With similar arguments as above, both undo log and intentions list can be used to cope with recovery. If ordered shared locks are not used, the current state is not required to construct the view of the operation, therefore storing the committed state in a stable storage and using intentions list to cope with recovery is more convenient. However, if ordered shared locks are used, both the committed state and the current state are required to construct the view and generate the response of an operation, thus using either recovery method does not have any implication on the implementation of the protocol.
6. Discussion

A concurrency control protocol, which uses context-specific information of executions, for atomic data types in distributed systems is presented. In general, the context of an execution depends only on the syntax of the execution and it is independent of the semantics of the transactions. In particular, the context-specific information used in our protocol is based on the views of operations. This approach can determine the conflict relations among a set of operations dynamically by using both the forward commutativity relation and the right backward commutativity relation. Unlike the state-based approach presented in [11] in which explicit semantics of a data object is incorporated in the protocol, the implementation of our protocol is independent of the semantics of the atomic data types. The required semantics of the data object is captured in the commutativity tables only; therefore, once the forward commutativity and right backward commutativity tables are deduced from the specification of the data types, the protocol can be readily applied on the data types without any modifications.

The novelty of this approach is that the conflict relations among operations depend not only on the operation types but also on the context of the operations. Also, we use a hybrid recovery scheme that is based on both deferred update (DU) and update-in-place (UIP). This results in greater flexibility in lock relation assignments than the traditional approaches that only employ a static conflict table and a recovery strategy that is based on either DU or UIP. We show that all the histories accepted either by DU or UIP are accepted by the proposed protocol even without using ordered shared relationship between locks. However, the converse is not true, since there exists a history (as shown in Section 3.1) accepted by the proposed protocol without using ordered shared relationship between locks but the history is neither accepted by DU nor by UIP. Moreover, ordered shared relationship between locks can be applied to execute conflicting operations to further increase the concurrency. The hierarchy of the histories accepted by the protocols is shown in Fig. 6.

---

**Fig. 6.** Hierarchy of the histories accepted by the protocols.
We would like to point out that due to the introduction of ordered shared relations between all the conflicting operations, no response of an invocation will be delayed. A transaction may be blocked only when it requests commitment. But cascading aborts may occur in the proposed protocol. However, by using the same idea of the recoverability relationship proposed in [2] for restricting the ordered shared relation to certain type of operations, cascaded aborts can be avoided. The recoverability relationship, however, is static in the sense that it is determined by the operation types, whereas the ordered shared relationship presented in this paper is context specific. For example, a withdraw operation may have ordered shared relations on some concurrent deposit operations but at the same time have shared relations on another concurrent deposit operations. Moreover, the commutativity relation used in [2] is invocation based which is more restrictive than forward commutativity relation and the right backward commutativity relation. In contrast, our proposed protocol employs both commutativity relations dynamically resulting in a more flexible scheme for determining conflict relations among operations.

When ordered shared locks are not used, the time complexity to construct the view of an operation is $O(n)$, where $n$ is the number of active operations in an object. When ordered shared locks are used, for the implementation described in Section 3.2, the time complexity to construct the view of an operation is $O(n^2)$. Although an extra overhead is required by the proposed protocol, concurrency can be improved significantly by reducing the possibility of conflicts. Note that when there is conflict, a transaction has to be delayed until the corresponding conflicting transaction has terminated. Therefore, the extra overhead is justified especially when the transactions are relatively long, since the penalty for a delay due to conflict is heavy. Moreover, in the case of distributed database systems, the execution time for a transaction is usually longer because most of the time is spent in messages passing. Therefore, the overhead for executing an operation is insignificant when compared to the time for messages passing.

References


