4

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-367

COMMUTATIVITY-BASED CONCURRENCY CONTROL FOR ABSTRACT DATA TYPES

William E. Weihl

August 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

ADA200979

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-367	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Commutativity-Based Concurrency Control for Abstract Data Types

12. PERSONAL AUTHOR(S)
Weihl, William E.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	1988 August	29

16. SUPPLEMENTARY NOTATION
(This document) The author

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
atomic transactions, concurrency control, locking, local atomicity, dynamic atomicity, abstract data types, recovery, undo logs intentions lists

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

~~We~~ presents two novel concurrency control algorithms for abstract data types. The algorithms ensure serializability of transactions by using conflict relations based on the commutativity of operations. ~~We~~ proves that both algorithms ensure a local atomicity property called dynamic atomicity. This means that the algorithms can be used in combination with each other and with other algorithms, as long as the other algorithms also ensure dynamic atomicity. (Dynamic atomic concurrency control algorithms include most two-phase locking algorithms, as well as some non-conflict-based algorithms and some optimistic algorithms.) The algorithms are quite general, permitting operations be be both partial and non-deterministic. In addition, the results returned by operations can be used in determining conflicts, thus permitting higher levels of concurrency than is otherwise possible. In contrast to most other work, our descriptions and proofs encompass recovery as well as concurrency control. Keywords; [illegible] systems, [illegible]

The two algorithms use different recovery methods: one uses intentions lists, and the other uses undo logs. We show that conflict relations that work with one (continued...)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE *(Include Area Code)* (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

19. recovery method do not necessarily work with the other, thus illustrating the subtle interactions between concurrency control and recovery. In addition, we identify a general correctness condition that must be satisfied by the combination of a recovery method and a conflict relation; we hope that this correctness condition will serve to simplify the analysis of other algorithms as well.

Commutativity-Based Concurrency Control for Abstract Data Types

William E. Weihl[1]

Abstract

We present two novel concurrency control algorithms for abstract data types. The algorithms ensure serializability of transactions by using conflict relations based on the commutativity of operations. We prove that both algorithms ensure a local atomicity property called *dynamic atomicity*. This means that the algorithms can be used in combination with each other and with other algorithms, as long as the other algorithms also ensure dynamic atomicity. (Dynamic atomic concurrency control algorithms include most two-phase locking algorithms, as well as some non-conflict-based algorithms and some optimistic algorithms.) The algorithms are quite general, permitting operations to be both partial and non-deterministic. In addition, the results returned by operations can be used in determining conflicts, thus permitting higher levels of concurrency than is otherwise possible. In contrast to most other work, our descriptions and proofs encompass recovery as well as concurrency control.

The two algorithms use different recovery methods: one uses intentions lists, and the other uses undo logs. We show that conflict relations that work with one recovery method do not necessarily work with the other, thus illustrating the subtle interactions between concurrency control and recovery. In addition, we identify a general correctness condition that must be satisfied by the combination of a recovery method and a conflict relation; we hope that this correctness condition will serve to simplify the analysis of other algorithms as well.

[1]Author's address: MIT Laboratory for Computer Science, Room 524, 545 Technology Square, Cambridge, MA 02139. E-mail (Arpanet): weihl@xx.lcs.mit.edu.

This research was supported in part by the National Science Foundation under Grants DCR-8510014 and CCR-8716884, in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, and in part by an IBM Faculty Development Award.

1. Introduction

Atomic transactions have been widely studied for over a decade as a mechanism for coping with concurrency and failures, particularly in distributed systems [14, 27, 8, 1]. A major area of research during this period has involved the design and analysis of concurrency control algorithms, for which an extensive theory has been developed (e.g., see [22, 3]). Initial work in the area left the data uninterpreted, or viewed operations as simple reads and writes. Recently, a number of researchers have considered placing more structure on the data accessed by transactions, and have shown how this structure can be used to permit more concurrency [11, 28, 29, 25, 1, 2, 19, 32, 31, 21]. For example, in our own work we have shown how the specifications of abstract data types can be used to permit high levels of concurrency [28, 29]. These techniques have been used in existing systems to deal with "hot-spots." In addition, such techniques are useful in general distributed systems, and may also prove useful in object-oriented database systems.

In this paper we present two new concurrency control algorithms for abstract data types. Both are locking algorithms: to execute an operation on an object, a transaction must acquire a lock on the object in a mode appropriate for the operation. Conflicts between lock modes for operations on an object are determined based on an analysis of the specification of the object's data type. Unlike algorithms based on reads and writes, the algorithms permit a high level of concurrency. Informally, two operations are required to conflict only if they do not "commute"; thus, concurrent transactions are allowed to update the same entity as long as the updates commute.

We describe our algorithms formally, and prove them correct. Our descriptions and proofs cover both concurrency control and recovery; indeed, as we discuss in more detail below, it is our belief that concurrency control and recovery interact in subtle ways, and that they need to be analyzed together. (Our analysis covers only recovery from aborts of transactions; we are working on developing a formal model for crashes and on analyzing crash recovery algorithms.)

We prove that both algorithms ensure *dynamic atomicity*, which is one of several *local atomicity properties* defined in [28, 29]. It is well-known that different "correct" concurrency control algorithms cannot be used in the same system and still ensure serializability. Local atomicity properties are interesting because different algorithms can be used in a single system, as long as the algorithms satisfy the same local atomicity property. Thus, by proving that our algorithms ensure dynamic atomicity, we show that they can be used in combination with a variety of other dynamic atomic concurrency control algorithms, including the non-locking algorithms in [28] and some optimistic algorithms [12]. The two algorithms themselves, even though they differ in the level of concurrency they support, can even be used in the same system. This is an important modularity result, since different algorithms may be most appropriate for different parts of a system.

Our algorithms generalize most known two-phase locking algorithms, such as those in [6, 11, 25, 2].[2] Unlike these other algorithms, both of our algorithms permit the *results* returned by an operation execution, as well as its name and arguments, to be used in determining the lock required by the operation. As illustrated in the body of the paper, this allows us to provide greater concurrency while still ensuring serializability, since the concurrency control algorithm has more information to work with. Some other algorithms (e.g., see [25]) achieve the effect of using information about results by acquiring a restrictive lock when an operation starts running, and then "down-grading" the lock depending on how the operation actually executes. The resulting protocol violates two-phase locking, and as a result *ad hoc* correctness arguments are usually given. Our algorithms show how results of operations, as well as names and arguments, can be used systematically.

[2]Our algorithms use *strict* two-phase locking; i.e., locks acquired by a transaction are not released until the end of the transaction. The algorithms in [6] permit locks to be released before the end of a transaction. The restriction to strict two-phase locking is useful for dealing with aborts, since it is not necessary to re-acquire locks to undo the effects of an aborting transaction.

Our algorithms are also quite general: they work for arbitrary data types, including ones with partial and non-deterministic operations. Most previous work deals only with total, deterministic operations. Examples of interesting data types with partial and non-deterministic operations can be found in [28, 29].

While an extensive theory has been developed for concurrency control, there has been less theoretical analysis of recovery algorithms, although some work does exist (e.g., see [9, 5]). One notable aspect of most work on concurrency control and recovery is that the two are treated as separate problems. Concurrency control is analyzed assuming that recovery is correct, and vice-versa. Indeed, it is almost a folk theorem in the field that a correct two-phase locking algorithm is obtained if operations are required to conflict whenever they do not commute. Unfortunately, as is well-known, concurrency control and recovery interact in subtle ways. This is particularly true for concurrency control algorithms that permit concurrent updates, such as the ones described in this paper.

For example, a "correct" recovery algorithm such as value logging [8, 24] does not work in combination with a concurrency control algorithm that permits concurrent updates. Consider an execution in which two concurrent transactions A and B modify the same entity X. Suppose A modifies X first, saving the initial value of X to be restored when A aborts. When B modifies X, it saves the value written by A. Now suppose B commits and then A aborts. The value saved by A will be restored, thus restoring X to its initial value and erasing both updates. More complex recovery algorithms, based on intentions lists or undo operations, have been designed for these more sophisticated concurrency control algorithms that permit concurrent updates. However, a theory of their interactions is sadly lacking.

The two algorithms presented in this paper use two different recovery algorithms: one uses intentions lists, and one uses undo logs. The fundamental difference between these two recovery algorithms lies in the "state" used to execute an operation for a transaction. Using intentions lists, an operation is executed for a transaction in a state that reflects the effects of the committed transactions, and of the transaction itself, but not of other active transactions. Using undo logs, an operation is executed in a state that reflects the effects of the committed transactions and *all* active transactions. As we show below, this difference between the two recovery algorithms has a subtle impact on concurrency control. In particular, while both algorithms use a conflict relation based on commutativity—two operations conflict if they do not "commute"—the precise definitions of commutativity needed are *different* for the two recovery algorithms. Thus, the truth of the folk theorem mentioned above depends in subtle ways on the precise definition of commutativity, and on the assumptions being made about recovery. Our descriptions of the algorithms are designed to highlight these assumptions.

This paper represents the start of an effort to develop a better understanding of the interactions between concurrency control and recovery. The examples in this paper illustrate that there is no single notion of correctness such that any "correct" concurrency control algorithm can be used with any "correct" recovery algorithm and guarantee that transactions are atomic. Our approach in this paper is formal in part because the interactions between concurrency control and recovery are very subtle. It is easy to be informal and wrong, or to avoid stating critical assumptions that are necessary for others to be able to build on the work.

One of the contributions of this paper is a correctness theorem (Theorem 14) that appears to capture the property that must be ensured by concurrency control and recovery *together*. (A similar invariant is used by O'Neil for a different algorithm [21].) We are currently studying other algorithms to see if this property applies to them as well.

The formal model used in this paper, and the definitions of atomicity and related properties, are taken from [28, 29]. We summarize the relevant material in Sections 2 and 3. Then, in Section 4, we define the two notions of commutativity used in the two algorithms. The two algorithms check for conflicts in essentially the same way, so in Section 5 we present a general conflict-based locking algorithm, ignoring recovery. Next, in Section 6, we describe how the two recovery techniques can be added to the general locking algorithm from Section 5, and prove the two algorithms correct. Finally, we conclude with a summary and discussion.

2. Model

Our model of computation is taken from [28, 29]; in this section we describe the details relevant to this paper. There are two kinds of entities in our model, *transactions* and *objects*. We assume that each object provides operations that can be called by transactions to examine and modify the object's state, and furthermore that these operations constitute the sole means by which transactions can access the state of the object. We will typically use the symbols A, B, and C for transactions, and X, Y, and Z for objects. We use ACT to denote the set of transactions.

Our model of computation is event-based, focusing on the events at the interface between transactions and objects. There are four kinds of events of interest:

- Invocation events, corresponding to the invocation of an operation by a transaction A at an object X, and denoted <inv,X,A>, where inv records the name of the operation *and* its arguments.
- Response events, corresponding to an object X returning a response to an operation invoked by a transaction A, and denoted <ret,X,A> where ret records the results of the operation.
- Commit events, corresponding to an object X learning that a transaction A has committed, and denoted <commit,X,A>.
- Abort events, corresponding to an object X learning that a transaction A has aborted, and denoted <abort,X,A>.

We refer to commit and abort events collectively as *completion* events. Each event indicates the transaction and object that participated in it; given an event <e,X,A>, we say that the event *involves* X and A.

We model a computation as a sequence of events. If H is a sequence of events and $\mathcal{X}$ is a set of objects, we define $H|\mathcal{X}$ ("H restricted to $\mathcal{X}$") to be the subsequence of H consisting of the events involving objects in $\mathcal{X}$; if $\mathcal{A}$ is a set of transactions, we define $H|\mathcal{A}$ similarly. If X is an object and A is a transaction, we will write H|X for H|{X}, and H|A for H|{A}. We define *committed*(H) to be the set of transactions that commit in H (i.e., for which a commit event occurs in H), and *aborted*(H) to be the set of transactions that abort in H. We also define *completed*(H) to be the set of transactions that complete (i.e., commit or abort) in H; that is, completed(H) = committed(H)$\cup$aborted(H).

We will use the following notation for sequences: the symbol "•" denotes concatenation of sequences, and the symbol "Λ" denotes the empty sequence.

Not all event sequences make sense as computations: each transaction is supposed to be a sequential process.[3] Thus, we require event sequences to be *well-formed*, as defined below. We refer to a well-formed event sequence as a *history*.

Well-formedness is a restriction on the individual transactions, not on their interactions at the objects. Thus, we define well-formedness first for sequences involving only a single transaction; an event sequence H is then well-formed if H|A is well-formed for every transaction A. H|A is well-formed if it satisfies the following conditions:

- The transaction A must wait for the response to one invocation before invoking another operation, and an object can generate a response for A only if A has a pending invocation. More precisely, let *op-events*(H|A) be the subsequence of H|A consisting of all invocation and response events; then op-events(H|A) must consist of an alternating sequence of invocation and response events, beginning with an invocation event. In addition, an invocation event and the immediately succeeding response

[3]We view concurrency within a transaction as something to be achieved with nested transactions [18, 23, 14], but the formal model used here does not include nesting. See [17] for a formal model of nested transactions.

event must involve the same object.

- The transaction A can commit or abort in H, but not both; i.e., committed(H|A) $\cap$ aborted(H|A) = $\emptyset$.[4]
- The transaction A cannot commit if it is waiting for the response to an invocation, and cannot invoke any operations after it commits. More precisely, if A $\in$ committed(H|A), then H|A consists of op-events(H|A) followed by some number of commit events, and op-events(H|A) ends in a response event.

These restrictions on transactions are intended to model the typical uses of transactions. A transaction executes by invoking operations on objects, receiving results when the operations finish. Since we disallow concurrency within a transaction, a transaction is permitted at most one pending invocation at any time. After receiving a response from all invocations, a transaction can commit at one or more objects.

We make very few restrictions on aborted transactions; for example, a transaction can continue to invoke operations after it has aborted. We have two reasons for avoiding additional restrictions. First, we have no need for them in our analysis. Second, and more important, additional restrictions might be too strong to model systems with orphans [20, 15, 10].

3. Global and Local Atomicity

In this section we define atomicity and several related properties. The definitions are taken from [28, 29]. Abstraction, and in particular data abstraction, plays an important role in these definitions. In particular, the definition of atomicity is based on the specifications of the objects in a system: transactions are atomic if their execution appears to be serializable and recoverable, as far as the transactions can tell given the specifications of the objects. For example, a system may be atomic at one level of abstraction and non-atomic at lower levels (cf. [2, 19]).

Since specifications play a central role in our definitions, we begin in Section 3.1 by describing our model of specifications for objects. Then, in Section 3.2, we define global atomicity. Next, in Section 3.3, we define dynamic atomicity, a *local atomicity property*. Dynamic atomicity, however, does not lend itself directly to inductive proofs. Thus, in Section 3.4 we define *online dynamic atomicity*, which is a strengthening of dynamic atomicity that captures the additional properties guaranteed by *pessimistic* concurrency control algorithms, and is more easily proved inductively.

3.1. Specifications

Our specifications take the form of sets of sequences. A set of sequences is a language, and can be conveniently described by an automaton. In addition, we will describe algorithms using automata, showing that all the sequences accepted by an automaton satisfy certain correctness conditions.

Atomicity, as defined in the next section, is based on a specification of how objects are permitted to behave in the absence of concurrency and failures. This specification, which we call the *serial specification* of an object, is quite like the specification of an ordinary data abstraction in a sequential, failure-free environment. In addition to its serial specification, an object will usually have a *behavioral specification*, which is a set of histories that characterizes the object's behavior in the face of failures and concurrent access by transactions. The distinction between the serial and behavioral specification of an object, which was introduced in [28, 29], is useful for structuring the design process. In addition, the serial specification of an object is typically much simpler than the behavioral specification, yet to reason about many properties of a transaction system it is often sufficient to use only

[4]This requirement, called *atomic commitment*, can be implemented using a commitment protocol such as two-phase commit [7, 13] or three-phase commit [26].

the serial specifications for objects. For the purposes of this paper, we will describe only the serial specifications; the details of the behavioral specifications are not important.

The serial specification of an object X is intended to capture the acceptable behavior of X in a sequential, failure-free environment. We could model the serial specification of X as a set of histories, where the histories satisfy certain restrictions (e.g., all transactions commit, and events of different transactions do not interleave). We have found it convenient, however, to use a slightly different model for serial specifications. Instead of a set of histories, we will use a set of *operation sequences*. An *operation* is a pair consisting of an invocation and a response to that invocation; in addition, an operation identifies the object on which it is executed. An operation does *not* identify a transaction; we have found no need to date for the serial specification of an object to vary depending on which transaction executes an operation, and indeed find it more convenient to describe serial specifications in a way that is independent of transactions.

We often speak informally of an "operation" on an object, as in "the insert operation on a set object." An operation in our formal model is intended to represent a single execution of an "operation" as used in the informal sense. For example, the following might be an operation (in the formal sense) on a set object X:

X:[insert(3),ok]

This operation represents an execution of the insert operation (in the informal sense) on X with argument "3" and result "ok."

As mentioned earlier, an automaton is a convenient tool for describing a set of sequences. In this paper we use a particular kind of automaton, called a *state machine*. A state machine consists of a (potentially infinite) set of states, a set of transitions, an initial state, and a partial transition function. The transition function maps (state,transition) pairs to states; if the transition function is defined on a given pair (s,t), we say that t *is defined in* s. The transition function can be extended in the obvious way to finite sequences of transitions. If T is a sequence of transitions and s is a state, we write T(s) for the value of the transition function on the pair (s,T). We use the notation $T(s) = \perp$ to indicate that T is not defined in s.

We say that a sequence of transitions is *accepted* by a state machine M if it is defined in the initial state of M. We define the *language* of a machine M, denoted L(M), to be the set of finite sequences of transitions that are accepted by M.

In this paper, we will describe transition functions by specifying the triples (s',t,s) such that t(s') = s. In each case it should be clear that for each s' and t there is at most one triple of the form (s',t,s), so the transition functions are well-defined. A more general model, permitting transition relations, is described in [16]; the greater generality is not needed for our purposes here.

From this point on, we assume that the serial specification of an object X is defined to be the language of a state machine S(X),[5] where the transitions of S(X) are the operations on X. We use $I_{S(X)}$ to denote the initial state of S(X). For example, consider an integer set object SET, initially empty, with operations to insert an element, to delete an element, and to test for membership. S(SET) is defined as follows. A state s of S(SET) is a set of integers; the initial state is the empty set. The transition function of S(SET) is defined by the preconditions and postconditions below; if op is an operation of SET, then op(s')=s if s' satisfies the precondition for op and s' and s satisfy the postcondition. We follow the convention here and throughout the paper that if a component of the state is not

[5]This assumption is not essential; all the definitions and results can be stated purely in terms of the serial specification as a set of operation sequences, without relying on a particular presentation in terms of a state machine. However, the use of a state machine makes the presentation somewhat more concrete, and hence (we hope) clearer.

mentioned in the postcondition, it is the same in s and s'; also, if no precondition is given, it is assumed to be the predicate "true".

SET:[insert(i),ok]
  Postcondition:
    $s = s' \cup \{i\}$

SET:[delete(i),ok]
  Postcondition:
    $s = s' - \{i\}$

SET:[member(i),true]
  Precondition:
    $i \in s'$

SET:[member(i),false]
  Precondition:
    $i \notin s'$

L(S(SET)) includes the following sequence of operations:

SET:[member(2),false]
SET:[member(3),false]
SET:[insert(3),ok]
SET:[member(3),true]

However, it does not include the following sequence:

SET:[member(2),true]
SET:[member(3),false]
SET:[insert(3),ok]
SET:[member(3),false]

The member operation returns true if and only if its argument has been inserted and not subsequently deleted; the first sequence above satisfies this constraint, while the second does not.

As an example of an object with partial and non-deterministic operations, we give the specification of a *semi-queue* object SEMIQ, adapted from [30]. A semi-queue is similar to a FIFO queue in that it provides operations to enqueue and dequeue items. However, it differs in that the operation to dequeue an item is non-deterministic: it is permitted to remove and return *any* item in the semi-queue. The non-determinism in the specification of the dequeue operation allows us to permit more concurrency among transactions using the semi-queue than we could permit among transactions using a FIFO queue. In addition, the dequeue operation is specified to be partial: if the semi-queue is empty, there is no possible result. In a concurrent setting, one could ensure atomicity relative to this specification by having an invocation of the dequeue operation block when the semi-queue is empty. In more detail, S(SEMIQ) is defined as follows. A state s of S(SEMIQ) is a multi-set (or bag) of items; the initial state is the empty bag. The transition function is defined by the preconditions and postconditions below:

SEMIQ:[enqueue(i),ok]
  Postcondition:
    $s = s' \cup \{i\}$

SEMIQ:[dequeue,i]
  Precondition:
    $i \in s'$
  Postcondition:
    $s = s' - \{i\}$

Enqueueing an item simply adds the item to the state; any item in the state can be returned by the dequeue operation, which also removes the item from the state.

Notice that restricting ourselves to transition functions, instead of relations, does not prevent us from specifying objects with partial and non-deterministic operations (in the informal sense). An invocation may have many possible results (or none) in a given state; each such result corresponds to a different operation (in the formal sense), and the transition function gives the new state for each operation.

3.2. Global Atomicity

Informally, a history of a system is atomic if the committed transactions in the history can be executed in some serial order and have the same effect. In our model, the permissible serial executions are characterized by the serial specifications of the objects. Thus, whether a history is atomic depends on these serial specifications. In this section we provide a formal definition of atomicity in terms of the serial specifications of the objects in a system.

Since serial specifications are sets of operation sequences, not sets of histories, we need to establish a correspondence between histories and operation sequences. We do this as follows. We say that a history is *serial* if events for different transactions are not interleaved. If H is a serial history, and A_1, ..., A_n are the transactions in H in the order in which they appear, then we can write H as $H|A_1 \bullet ... \bullet H|A_n$. We say that a history H is *failure-free* if aborted(H) = ∅. Now, if H is a serial failure-free history, we define *OpSeq*(H) (the operation sequence corresponding to H) as follows. For a transaction A, OpSeq(H|A) is the operation sequence obtained from H|A by pairing each invocation event with its corresponding termination event, and discarding commit events and pending invocation events. Let A_1, ..., A_n be the transactions in H in the order in which they appear; then OpSeq(H) is defined to be $\text{OpSeq}(H|A_1) \bullet ... \bullet \text{OpSeq}(H|A_n)$.

For example, if H is the serial failure-free history

<insert(3),SET,B>
<ok,SET,B>
<commit,SET,B>
<member(3),SET,A>
<true,SET,A>
<commit,SET,A>

then OpSeq(H) is the operation sequence

SET:[insert(3), ok]
SET:[member(3), true]

We say that a serial failure-free history H is *acceptable at X* if OpSeq(H|X) is an element of the serial specification of X; in other words, if the sequence of operations in H involving X is permitted by the serial specification of X. A serial failure-free history is *acceptable* if it is acceptable at every object X.

Equivalently, given our assumption that the serial specification of X is defined as the language of a state machine S(X), then a serial failure-free history H is acceptable at X if OpSeq(H|X) is defined in $I_{S(X)}$.

We say that two histories H and K are *equivalent* if every transaction performs the same steps in H as in K; i.e., if H|A = K|A for every transaction A. If H is a history and T is a total order on transactions, we define *Serial*(H,T) to be the serial history equivalent to H in which transactions appear in the order T. Thus, if A_1, ..., A_n are the transactions in H in the order T, then $\text{Serial}(H,T) = H|A_1 \bullet ... \bullet H|A_n$.

If T is a total ordering of transactions, we then say that a failure-free history H is *serializable in the order T* if Serial(H,T) is acceptable. In other words, H is serializable in the order T if, according to the serial specifications of the objects, it is permissible for the transactions in H, when run in the order T, to execute the same steps as in H. We say that a failure-free history H is *serializable* if there exists a total order T on transactions such that H is serializable in the order T.

Now, define *permanent*(H) to be H|committed(H). We then say that H is *atomic* if permanent(H) is serializable. Thus, we formalize recoverability by throwing away events for non-committed transactions, and requiring that the committed transactions be serializable.

3.3. Local Atomicity

The definition of atomicity given above is global: it applies to a history of an entire system. To build systems in a modular, extensible fashion, it is important to define local properties of objects that guarantee a desired global property such as atomicity. In this section we define a particular *local atomicity property*, which we call *dynamic atomicity*. A *local atomicity property* is a property ***P*** of specifications of objects such that the following is true: if the specification of every object in a system satisfies ***P***, then every history in the system's behavior is atomic. As an aside, we remark that dynamic atomicity is an *optimal* local atomicity property: no strictly weaker local property suffices to ensure global atomicity [28, 29].

The problem that must be solved in designing a local atomicity property is to ensure that the objects agree on at least one serialization order for the committed transactions. Solving this problem can be difficult because each object is aware of only the events in which it participates. In other words, each object has purely *local* information; no object has complete information about the *global* computation of the system. As illustrated in [28, 29], if different objects use "correct" but incompatible concurrency control methods, non-serializable executions can result. A local atomicity property describes how objects agree on a serialization order for committed transactions.

We will define dynamic atomicity as a property of the local history at an individual object, in such a way that if the local history at each object is dynamic atomic, then the history of the entire system will be atomic. An object is then said to be dynamic atomic if every history permitted by its behavioral specification is dynamic atomic.

Most concurrency control algorithms, including two-phase locking [6, 4, 11], determine a serialization order for transactions *dynamically*, based on the order in which transactions invoke operations and obtain locks on objects. Dynamic atomicity characterizes the behavior of algorithms that are dynamic in this sense. Informally stated, the fundamental property of protocols characterized by dynamic atomicity is the following: if the sequence of operations executed by one committed transaction conflicts with the operations executed by another committed transaction, then some of the operations executed by one of the transactions must occur after the other transaction has committed. In other words, if two transactions are completely concurrent (neither executes an operation after the other commits), they must not conflict. Locking protocols (and all pessimistic protocols) achieve this property by *delaying* conflicting operations; optimistic protocols [12] achieve this property by allowing conflicts to occur, but *aborting* conflicting transactions to prevent conflicts among committed transactions.

We can describe dynamic atomicity precisely as follows. If H is a history, define *precedes*(H) to be the following relation on transactions: (A,B)$\in$precedes(H) if and only if there exists an operation invoked by B that responds after A commits in H. The events need not occur at the same object. The relation precedes(H) captures the concept of one transaction occurring after another: if (A,B)$\in$precedes(H), then some operation executed by B occurred in H after A committed. This could have happened because B started after A finished or ran more slowly than A, or because B was delayed because of a conflict with A.

The following lemma from [28, 29] provides the key to our definition of dynamic atomicity.

Lemma 1: If H is a history and X is an object, then precedes(H|X) $\subseteq$ precedes(H).

If H is a history of the system, each object has only partial information about precedes(H). However, if each object

X ensures local serializability in *all* orders consistent with precedes(H|X), by the lemma we are guaranteed global serializability in all orders consistent with precedes(H). (It is easy to show that precedes(H) is a partial order if H is well-formed, so such orders exist.) To be precise, we have the following definition of dynamic atomicity: we say that a history H is *dynamic atomic* if permanent(H) is serializable in every total order consistent with precedes(H). In other words, every serial history equivalent to permanent(H), with the transactions in an order consistent with precedes(H), must be acceptable.

The following theorem, taken from [28, 29], justifies our claim that dynamic atomicity is a local atomicity property:

Theorem 2: If every local history in the behavioral specification of each object in a system is dynamic atomic, then every history in the system's behavior is atomic.

3.4. Online Dynamic Atomicity

Dynamic atomicity is ensured by many algorithms, including most pessimistic algorithms and some optimistic ones. The algorithms to be presented later in this paper are pessimistic. To prove them correct, it is useful to strengthen dynamic atomicity in two ways. First, dynamic atomicity does not constrain the active transactions. However, pessimistic concurrency control algorithms have the property that once a transaction executes an operation, it can commit at any time. Second, dynamic atomicity requires the permanent part of a history H to be serializable in all orders consistent with precedes(H); it is possible, however, for the different orders to be distinguishable by later transactions. (For example, if two concurrent transactions enqueue items on a queue and commit, they are serializable in both possible orders, since it is always possible to enqueue. Later transactions, however, care about the order, and hence will never be able to dequeue.) Most practical concurrency control and recovery algorithms retain only a single "state" that represents the effects of the committed transactions. To ensure that we can schedule later transactions by looking only at this state, and not at more detailed history information, we need to ensure that all required serializations of the committed transactions are equivalent in the sense that they are indistinguishable by later transactions (i.e., they result in the same final state).

In this section we define *online dynamic atomicity*, which strengthens dynamic atomicity to avoid these two problems. First, if H is a history and C is a set of transactions, we say that C is a *commit set* for H if committed(H) $\subseteq$ C and C $\cap$ aborted(H) = $\emptyset$. In other words, C is a set of transactions that have already committed or might commit. We then say that a history H is *online dynamic atomic* at X if the following conditions are satisfied for every commit set C for H:

- H|X|C is serializable in every total order consistent with precedes(H|X).
- if T and U are total orders consistent with precedes(H|X), then OpSeq(Serial(H|X|C,T))($I_{S(X)}$) = OpSeq(Serial(H|X|C,U))($I_{S(X)}$).

In other words, regardless of which active transactions commit, the resulting history will be dynamic atomic, and furthermore the different serialization orders cannot be distinguished by future transactions. The first constraint is necessary for a pessimistic algorithm to ensure dynamic atomicity; the second constraint allows an algorithm to summarize the effects of the committed transactions with a single "state."

We say that H is online dynamic atomic if, for all objects X, it is online dynamic atomic at X. The following lemma is immediate:

Lemma 3: If H is online dynamic atomic, H is also dynamic atomic.

Online dynamic atomicity seems to be a fundamental invariant preserved by pessimistic algorithms that guarantee dynamic atomicity. We will demonstrate the correctness of the algorithms presented later by showing that they guarantee online dynamic atomicity.

4. Commutativity

Both algorithms described later in this paper use conflict relations based on "commutativity:" two operations conflict if they do not "commute." However, the different recovery algorithms require subtly different notions of commutativity. In this section we describe the two definitions and give some examples to illustrate how they differ.

It is important to point out that we define the two notions of commutativity as binary relations on operations in the sense of our formal definition, rather than simply for invocations as is usually done. Thus, the locks acquired by an operation can depend on the results returned by the operation. In addition, it is convenient to phrase our definitions in terms of sequences of operations, not just individual operations. The definitions below are given in terms of general state machines; they are applied in subsequent sections to the serial specifications of objects.

4.1. Forward Commutativity

If M is a state machine, and R and S are two sequences of transitions of M, we say that R and S *commute forward* if, for every state s in which R and S are both defined, R(S(s)) = S(R(s)) and R(S(s)) $\neq \perp$. The motivation for our choice of terminology is the following: if R and S are both defined in a state s, we can "push R forward over S" (and vice-versa) to show that R is defined in S(s), S is defined in R(s), and the final states are the same in the two cases. Notice that the forward commutativity relation is symmetric.

For example, consider the set object SET whose serial specification was described earlier in Section 3.1. The forward commutativity relation on operations of SET is described by the table in Figure 4-1. For example, the insert operation commutes forward with the member operation if they involve different elements, or if the member operation returns true; similarly, the delete operation commutes forward with the member operation if they involve different elements, or if the member operation returns false. The insert and delete operations commute forward only if they involve different elements.

	SET:[insert(j),ok]	SET:[delete(j),ok]	SET:[member(j),true]	SET:[member(j),false]
SET:[insert(i),ok]		i=j		i=j
SET:[delete(i),ok]	i=j		i=j	
SET:[member(i),true]		i=j		
SET:[member(i),false]	i=j			

An entry indicates that the operations for the given row and column do not commute forward if the indicated condition is true.

Figure 4-1: Forward Commutativity Relation for SET

As another example, consider a bank account object BA, with operations to deposit and withdraw money, and to retrieve the current balance. Assume that a withdrawal has two possible results, OK and NO. We take the serial specification of BA to be the language of a machine S(BA), which is defined as follows. A state s of S(BA) is a positive integer; the initial state is 0. The transition function of S(BA) is defined by the preconditions and postconditions below:

BA:[deposit(i),ok], i>0
 Postcondition:
  $s = s'+i$

BA:[withdraw(i),OK], i>0
 Precondition:
  $s' \geq i$
 Postcondition:
  $s = s'-i$

BA:[withdraw(i),NO], i>0
 Precondition:
  $s' < i$

BA:[balance,i]
 Precondition:
  $s' = i$

The forward commutativity relation on operations of BA is given by the table in Figure 4-2. Deposits and successful withdrawals do not commute with balance operations, since the former change the state. Similarly, successful withdrawals do not commute with each other; for example, BA:[withdraw(i),OK] and BA:[withdraw(j),OK] are both defined in any state $s \geq \max(i,j)$, but if $s<i+j$ then the two operations cannot be executed in sequence starting in state s.

	BA:[deposit(j),ok]	BA:[withdraw(j),OK]	BA:[withdraw(j),NO]	BA:[balance,j]
BA:[deposit(i),ok]			×	×
BA:[withdraw(i),OK]		×		×
BA:[withdraw(i),NO]	×			
BA:[balance,i]	×	×		

× indicates that the operations for the given row and column do not commute forward.

Figure 4-2: Forward Commutativity Relation for BA

We include here some lemmas about forward commutativity that will be useful later on.

The first lemma provides an inductive technique for proving that two sequences commute forward:

Lemma 4: Let R_i and S_j, for $i,j \in \{0,1\}$, be sequences of transitions. If R_i commutes forward with S_j for all i and j, then $R_0 \bullet R_1$ commutes forward with $S_0 \bullet S_1$.

Proof: Straightforward from the definitions. □

The following corollary extends the lemma to sequences composed of more than two parts.

Corollary 5: Let R_i, $1 \leq i \leq m$, and S_j, $1 \leq j \leq n$, be sequences of transitions, and let $R = R_1 \bullet ... \bullet R_m$ and $S = S_1 \bullet ... \bullet S_n$. If R_i commutes forward with S_j for all i and j, then R commutes forward with S.

The following lemma addresses the situation when we have a collection of more than two sequences that commute forward pairwise.

Lemma 6: Let s be a state, and let S_i, $1 \leq i \leq n$, be sequences of transitions such that:

1. S_i is defined in s for all i.

2. S_i commutes forward with S_j for all i and j, $1 \le i < j \le n$.

Let $i_1, i_2, \ldots, i_n$ be a permutation of $1, 2, \ldots, n$. Then:

1. $S_1 \bullet S_2 \bullet \ldots \bullet S_n(s) \neq \bot$

2. $S_{i_1} \bullet S_{i_2} \bullet \ldots \bullet S_{i_n}(s) = S_1 \bullet S_2 \bullet \ldots \bullet S_n(s)$

Proof: The proof proceeds by induction on n. The case when n=1 is trivial. The case when n=2 follows directly from the definition of forward commutativity.

For the induction step, assume that the lemma holds for fewer than n sequences. Let j be such that $i_j = 1$ (so $S_{i_j} = S_1$). By the induction hypothesis, $S_2 \bullet S_3 \bullet \ldots \bullet S_n(s) \neq \bot$, and $S_{i_1} \bullet \ldots \bullet S_{i_{j-1}} \bullet S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(s) = S_2 \bullet \ldots \bullet S_n(s)$.

Now let $s_1 = S_{i_1} \bullet \ldots \bullet S_{i_{j-1}}(s)$. Since $S_{i_1} \bullet \ldots \bullet S_{i_{j-1}} \bullet S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(s) \neq \bot$, it follows that $s_1 \neq \bot$ and $S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}$ is defined in s_1.

By Corollary 5, S_1 commutes forward with $S_2 \bullet \ldots \bullet S_n$, $S_{i_1} \bullet \ldots \bullet S_{i_{j-1}}$, and $S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}$. Since S_1 is defined in s, and so is $S_2 \bullet \ldots \bullet S_n$, it follows from the definitions that $S_2 \bullet \ldots \bullet S_n(S_1(s)) \neq \bot$. This proves the first half of the lemma.[6]

Now, since S_1 is defined in s, and so is $S_{i_1} \bullet \ldots \bullet S_{i_{j-1}}$, it follows from the definition of forward commutativity that S_1 is defined in s_1. By the commutativity relationships argued in the previous paragraph and the definition of forward commutativity, $S_1(S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(s_1)) = S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(S_1(s_1))$.

The following equalities show the second half of the lemma:

$$\begin{aligned}
S_1 \bullet S_2 \bullet \ldots \bullet S_n(s) &= S_2 \bullet \ldots \bullet S_n(S_1(s)) \\
&= S_1(S_2 \bullet \ldots \bullet S_n(s)) \\
&= S_1(S_{i_1} \bullet \ldots \bullet S_{i_{j-1}} \bullet S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(s)) \\
&= S_1(S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(s_1)) \\
&= S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(S_1(s_1)) \\
&= S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(S_1(S_{i_1} \bullet \ldots \bullet S_{i_{j-1}}(s)) \\
&= S_{i_{j+1}} \bullet \ldots \bullet S_{i_n}(S_{i_j}(S_{i_1} \bullet \ldots \bullet S_{i_{j-1}}(s)) \\
&= S_{i_1} \bullet S_{i_2} \bullet \ldots \bullet S_{i_n}(s).
\end{aligned}$$

The second line follows since S_1 commutes forward with $S_2 \bullet \ldots \bullet S_n$. The remaining lines follow from equalities argued above. □

4.2. Backward Commutativity

If M is a state machine, and R and S are two sequences of transitions of M, we say that R and S *commute backward* if R(S(s)) = S(R(s)) for every state s. The motivation for this terminology is that we will apply the definition in situations in which we know that $R(S(s)) \neq \bot$; from the definition we can "push R backward over S" to show that $S(R(s)) \neq \bot$, and that R(S(s)) = S(R(s)). The backward commutativity relation, like the forward commutativity relation, is symmetric.

The backward commutativity relations for sets and bank accounts are described in Figures 4-3 and 4-4. For example, let P=SET:[insert(i),ok] and let Q=SET:[member(i),true]. Notice that $P(s) = s \cup \{i\}$, while Q(s) is s if $i \in s$,

[6]Notice that "•" denotes concatenation, not functional composition, so $S_1 \bullet S_2 \bullet \ldots \bullet S_n(s) = S_2 \bullet \ldots \bullet S_n(S_1(s))$.

and is $\perp$ otherwise. Thus, P(Q(s)) is s if $i \in s$, and is $\perp$ otherwise, while Q(P(s)) = $s \cup \{i\}$. Hence P and Q do not commute backward. In other words, knowing that we can execute P and then Q does not tell us that we can execute Q and then P. Similarly, if P=BA:[deposit(i),ok] and Q=BA:[withdraw(j),OK], then P(s) = s+i, and Q(s) is s-j if $s \geq j$, and is $\perp$ otherwise. Thus, P(Q(s)) is s-j+i if $s \geq j$, and is $\perp$ otherwise, while Q(P(s)) is s+i-j (=s-j+i) if $s+i \geq j$, and is $\perp$ otherwise. *Hence, while P•Q and Q•P give the same result when both are defined, they are not defined over the same domain, and thus P and Q do not commute backward.*

	SET:[insert(j),ok]	SET:[delete(j),ok]	SET:[member(j),true]	SET:[member(j),false]
SET:[insert(i),ok]		i=j	i=j	i=j
SET:[delete(i),ok]	i=j		i=j	i=j
SET:[member(i),true]	i=j	i=j		
SET:[member(i),false]	i=j	i=j		

An entry indicates that the operations for the given row and column do <u>not</u> commute backward if the indicated condition is true.

Figure 4-3: Backward Commutativity Relation for SET

	BA:[deposit(j),ok]	BA:[withdraw(j),OK]	BA:[withdraw(j),NO]	BA:[balance,j]
BA:[deposit(i),ok]		×	×	×
BA:[withdraw(i),OK]	×		×	×
BA:[withdraw(i),NO]	×	×		
BA:[balance,i]	×	×		

× *indicates that the operations for the given row and column do <u>not</u> commute backward.*

Figure 4-4: Backward Commutativity Relation for BA

The rather subtle differences between the two notions of commutativity are shown by comparing Figures 4-3 and 4-4 to Figures 4-1 and 4-2. Interestingly, for the object SET, the forward commutativity relation is strictly larger than the backward commutativity relation: more operations commute forward than commute backward. For the object BA, however, the two relations are incomparable.

We now present a series of lemmas about backward commutativity that will prove useful later. The first lemma provides an inductive technique for showing that two sequences commute backward.

Lemma 7: Let R_i and S_j, for $i,j \in \{0,1\}$, be sequences of transitions. If R_i commutes backward with S_j for all i and j, then $R_0 \bullet R_1$ commutes backward with $S_0 \bullet S_1$.

Proof: If s is a state, we need to show that $R_0 \bullet R_1 \bullet S_0 \bullet S_1(s) = S_0 \bullet S_1 \bullet R_0 \bullet R_1(s)$. The following equalities follow from the definition of backward commutativity, and show the desired result:

$$\begin{aligned} R_0 \bullet R_1 \bullet S_0 \bullet S_1(s) &= S_1(S_0(R_1(R_0(s)))) \\ &= S_1(R_1(S_0(R_0(s)))) \\ &= S_1(R_1(R_0(S_0(s)))) \\ &= R_1(S_1(R_0(S_0(s)))) \end{aligned}$$

$$= R_1(R_0(S_1(S_0(s))))$$
$$= S_0\bullet S_1\bullet R_0\bullet R_1(s)$$

□

The following corollary extends the lemma to sequences composed of more than two parts.

Corollary 8: Let R_i, $1 \le i \le m$, and S_j, $1 \le j \le n$, be sequences of transitions, and let $R = R_1\bullet\ldots\bullet R_m$ and $S = S_1\bullet\ldots\bullet S_n$. If R_i commutes backward with S_j for all i and j, then R commutes backward with S.

The final lemma handles the case when there are more than two sequences. Unlike the corresponding lemma for forward commutativity, it does not require that all pairs of sequences commute backward. Instead, we can supply a partial order on the sequences, with the constraint that sequences unrelated by the partial order commute backward. The lemma then shows that if we pick a total order consistent with the partial order, and concatenate the sequences in the given total order, the function on the state corresponding to the concatenation of the sequences does not depend on the particular choice of total order. This result will be quite useful in our correctness proof of the undo log algorithm.

Lemma 9: Let s be a state, and let S_i be sequences of operations, for $1 \le i \le n$. Let P be a partial order on the integers such that if $(i,j) \notin P$ and $(j,i) \notin P$ then S_i commutes backward with S_j. Let T and U be total orders on the integers consistent with P, let $i_1, \ldots, i_n$ be the integers from 1 to n inclusive ordered by T, and let $j_1, \ldots, j_n$ be the integers from 1 to n inclusive ordered by U. Then

$$S_{i_1}\bullet\ldots\bullet S_{i_n}(s) = S_{j_1}\bullet\ldots\bullet S_{j_n}(s).$$

Proof: The proof is by induction on n. The case when n=1 is trivial. The case when n=2 follows directly from the definition of backward commutativity.

For the induction step, assume that the lemma holds for fewer than n sequences. Let k be such that $j_k = i_n$. By induction the lemma holds for the n-1 sequences $S_{i_1}, \ldots, S_{i_{n-1}}$. Therefore, $S_{i_1}\bullet\ldots\bullet S_{i_{n-1}}(s) = S_{j_1}\bullet\ldots\bullet S_{j_{k-1}}\bullet S_{j_{k+1}}\bullet\ldots\bullet S_{j_n}(s)$. Furthermore, since U orders $j_{k+1}, \ldots, j_n$ after j_k $(=i_n)$, T orders i_n after them, and T and U are both consistent with P, S_{i_n} must commute backward with $S_{j_{k+1}}$, $S_{j_{k+2}}$, ..., and S_{j_n}. Therefore, by Corollary 8, S_{i_n} commutes backward with $S_{j_{k+1}}\bullet\ldots\bullet S_{j_n}$. The following equations complete the proof:

$$S_{i_1}\bullet\ldots\bullet S_{i_n}(s) = S_{i_n}(S_{i_1}\bullet\ldots\bullet S_{i_{n-1}}(s))$$
$$= S_{i_n}(S_{j_1}\bullet\ldots\bullet S_{j_{k-1}}\bullet S_{j_{k+1}}\bullet\ldots\bullet S_{j_n}(s))$$
$$= S_{i_n}(S_{j_{k+1}}\bullet\ldots\bullet S_{j_n}(S_{j_1}\bullet\ldots\bullet S_{j_{k-1}}(s)))$$
$$= S_{j_{k+1}}\bullet\ldots\bullet S_{j_n}(S_{i_n}(S_{j_1}\bullet\ldots\bullet S_{j_{k-1}}(s)))$$
$$= S_{j_1}\bullet\ldots\bullet S_{j_{k-1}}\bullet S_{i_n}\bullet S_{j_{k+1}}\bullet\ldots\bullet S_{j_n}(s)$$
$$= S_{j_1}\bullet\ldots\bullet S_{j_n}(s)$$

□

5. Conflict-based Locking

The two algorithms to be presented in this paper are quite similar. They differ primarily in how they perform recovery, and in the particular conflict relations used for concurrency control. To exploit this common structure in our correctness proofs, and to highlight the differences between the two algorithms, we describe in this section a general conflict-based locking algorithm, and prove some properties about its behavior. Our description here covers only concurrency control, and not recovery. In addition, the locking algorithm described in this section uses an arbitrary conflict relation to decide when two locks conflict. In the next section, we show how particular conflict relations, based on the two notions of commutativity defined earlier, can be used with particular recovery techniques to ensure online dynamic atomicity. Like most algorithms based on two-phase locking, the algorithms described here are subject to deadlock; the usual remedies (e.g., timeout or detection) can be used to resolve deadlocks when they occur or to avoid them.

A locking concurrency control algorithm must keep track of the locks held by each transaction, as well as which transactions have committed and which have aborted. In addition, a response to an invocation should be generated for a transaction only if the transaction has a pending invocation; thus, we need to keep track of the pending invocations. Finally, since the two algorithms described in the next section maintain similar information about the "committed state" of the object, we describe how that information is maintained as part of the algorithm described in this section. In part, this allows us to prove a general theorem that shows what must be guaranteed by concurrency control and recovery together to ensure online dynamic atomicity; we then prove the correctness of the two algorithms in the next section by showing that they ensure this property.

Given an object X with its serial specification described by a state machine S(X), the conflict-based locking algorithm is defined by a state machine CBL(X). CBL(X) has a parameter CONFLICT, which is a binary relation on the operations of X. We assume that CONFLICT is a symmetric relation, and say that O_1 *conflicts with* O_2 if $(O_1,O_2) \in$ CONFLICT.

The transitions of CBL(X) are the events involving X. A state s of CBL(X) has five components, denoted s.permanent, s.pending, s.log, s.committed, and s.aborted. s.permanent is either $\perp$ or a state of S(X), used to keep track of the state resulting from the execution of the committed transactions. s.pending is a partial mapping from transactions to invocations; it is intended to record the pending invocation (if any) for each transaction. s.log is a mapping from transactions to sequences of operations; it is intended to record the sequence of operations executed by each transaction. s.committed and s.aborted are sets of transactions; they record the committed and aborted transactions, respectively. In the initial state s_0 of CBL(X), s_0.permanent is the initial state $I_{S(X)}$ of S(X), s_0.pending is undefined everywhere, s_0.log maps each transaction to the empty sequence, and s_0.committed and s_0.aborted are both the empty set.

The transition function of the state machine CBL(X) is defined by the preconditions and postconditions below. The notation $m[x \rightarrow y]$, where m is a (possibly partial) mapping, denotes the mapping that is identical to m except on x, which it maps to y.

```
<commit, X, A>
  Postcondition:
      s.committed = s'.committed ∪ {A}
      if A ∉ s'.committed then s.permanent = s'.log(A)(s'.permanent)

<abort, X, A>
  Postcondition:
      s.aborted = s'.aborted ∪ {A}

<inv, X, A>
  Postcondition:
      s.pending = s'.pending[A→inv]

<ret, X, A>
  Precondition:
      s'.pending(A) ≠ ⊥
      new-op = X:[s'.pending(A),ret]
      new-log = s'.log(A) • new-op
      ∀ B ∉ s'.committed ∪ s'.aborted ∪ {A}
          ∀ op ∈ s'.log(B)
              (new-op, op) ∉ CONFLICT
    Postcondition:
      s.pending = s'.pending[A→⊥]
      s.log = s'.log[A→new-log]
```

In this conflict-based locking algorithm, the set of lock modes is taken to be the set of operations (in our formal sense) on X. Conflicts between lock modes are defined by the relation CONFLICT. To return a response to an invocation, a lock must be acquired in the mode appropriate for the operation. Thus, to return a response ret to an invocation inv for transaction A, A must first acquire a lock on X in mode X:[inv,ret]. The set of locks held by an active (i.e., not completed) transaction is represented by the set of operations in its log. Thus, the precondition on response events requires that no other active transaction hold a lock that conflicts with the lock needed for the given response.

The rest of the description of the algorithm is straightforward. We view invocation and completion events as controlled by the transactions, so their preconditions are true, indicating that they can always happen.[7] Invocation events simply record the pending invocation. Similarly, commit and abort events simply record that the transaction has committed or aborted; in addition, commit events update the permanent component of the state by applying the committing transaction's log to it (assuming this is the first time the object has learned that the transaction has committed).

The remaining preconditions on response events simply require that the transaction have a pending invocation. The postcondition for a response event erases the record of the pending invocation, and appends the operation to the transaction's log.

Notice that we have not described how an object can tell what response events make sense. We have described some preconditions on response events — in particular, the transaction must have a pending invocation, and the operation formed by pairing the pending invocation and the response event in question must not conflict with any

[7]In describing the algorithm, we focus on what is controlled by the object, so we assume that events controlled by the transactions can always happen. In analyzing the algorithm later, we will only concern ourselves with histories — i.e., well-formed event sequences — permitted by the algorithm. The well-formedness constraints could be described as additional preconditions on the events in the description of the algorithm. This would ensure that all event sequences in the language of CBL(X) are well-formed. Adding the additional preconditions would be straightforward, and would simply clutter the description of the algorithm. It seems more convenient to deal with them as we have here.

operations already executed by other active transactions. As described, however, any response event that satisfies these two constraints could be produced by CBL(X). Additional constraints are needed for the algorithm to be correct; for example, if a transaction writes a value into X and then tries to read the value, it should see the value it wrote. In the next section, we will describe how a response event is determined by "executing" the pending invocation against a "state" of the object. The reason we have not done so here is that the two recovery algorithms described in the next section differ in the state used to execute a pending invocation.

It is worth noting that a practical implementation would not maintain the committed and aborted sets forever, or even do so explicitly. Similarly, the log would not be maintained forever. We could describe the algorithm in a way that avoided this seeming inefficiency, but the resulting description would be more complex and harder to analyze. It is not hard to show that an efficient implementation that discards such information implements the algorithms as they are described here.

We now present a series of lemmas, culminating in a theorem that provides us with a "correctness condition" that a recovery algorithm and concurrency control algorithm must satisfy together. The first lemma states some simple invariants relating a history in L(CBL(X)) to the state in which it leaves the machine.

Lemma 10: Let H be a history in L(CBL(X)), let s be the state of CBL(X) after H, and let A be a transaction. Then:

1. OpSeq(H|A) = s.log(A)
2. op-events(H|A) ends in an invocation event <inv,X,A> iff s.pending(A) = inv
3. committed(H) = s.committed
4. aborted(H) = s.aborted

Proof: Straightforward by induction on the length of H. □

The next lemma relates the "permanent" component of the state of CBL(X) to the operations executed by committed transactions, showing that s.permanent is the result of executing the committed transactions, starting in the initial state of S(X), in the order in which they commit. First, if H is a history, define *commit-order*(H) to be the partial order on transactions containing all pairs (A,B) such that A and B both commit in H and the first commit event for A appears before the first commit event for B.

Lemma 11: Let H be a history in L(CBL(X)), let s be the state of CBL(X) after H, and let T be a total order on transactions consistent with commit-order(H). Then s.permanent = OpSeq(Serial(H|committed(H),T))($I_{S(X)}$).

Proof: The proof is by induction on the length of H. The basis case, when H = Λ, is trivial. For the inductive step, suppose H = K•e, where e is a single event, and assume that the lemma holds for K. Let s_K be the state of CBL(X) after K. Notice that commit-order(K) ⊆ commit-order(H), so T is also consistent with commit-order(K). Since K is shorter than H, it follows from the induction hypothesis that s_K.permanent = OpSeq(Serial(K|committed(K),T))($I_{S(X)}$). There are now two cases.

First, if e is an invocation, response, or abort event, then committed(H) = committed(K), commit-order(H) = commit-order(K), H|committed(H) = K|committed(K), and s.permanent = s_K.permanent. The result now follows from the induction hypothesis.

Second, suppose e = <commit,X,A>. There are two subcases. If A∈committed(K), then committed(H) = committed(K), commit-order(H) = commit-order(K), H|committed(H) = K|committed(K), and s.permanent = s_K.permanent. As above, the result now follows from the induction hypothesis.

Otherwise, assume A∉committed(K). Then committed(H) = committed(K) ∪ {A}, commit-order(H) = commit-order(K) ∪ (committed(K) × {A}), and s.permanent = s_K.log(A)(s_K.permanent). Furthermore, OpSeq(H|A) = OpSeq(K|A). Since commit-order(H) orders A after everything in committed(K), and T is consistent with commit-order(H),

$$\begin{aligned} \mathrm{OpSeq(Serial(H|committed(H),T))} &= \mathrm{OpSeq(Serial(K|committed(H),T))} \\ &= \mathrm{OpSeq(Serial(K|committed(K),T))} \bullet \mathrm{OpSeq(K|A)} \end{aligned}$$

Thus,

$$\begin{aligned} &\mathrm{OpSeq(Serial(H|committed(H),T))}\ (I_{S(X)}) \\ &= [\ \mathrm{OpSeq(Serial(K|committed(K),T))} \bullet \mathrm{OpSeq(K|A)}\ ]\ (I_{S(X)}) \\ &= \mathrm{OpSeq(K|A)}\ (\ \mathrm{OpSeq(Serial(K|committed(K),T))}\ (I_{S(X)})\ ) \\ &= \mathrm{OpSeq(K|A)}\ (s_K.\mathrm{permanent}), && \text{by the induction hypothesis} \\ &= s_K.\mathrm{log(A)}\ (s_K.\mathrm{permanent}), && \text{by Lemma 10} \\ &= \mathrm{s.permanent}, && \text{by the definition of CBL(X)} \end{aligned}$$

This completes the proof. □

The next lemma states a simple invariant guaranteed by the test for conflicts in the precondition for response events: operations executed by distinct active transactions do not conflict. The proof relies on the assumption that the conflict relation is symmetric.

Lemma 12: Let H be a history in L(CBL(X)), let s be the state of CBL(X) after H, and let A and B be transactions such that $A \neq B$, $A \notin$ completed(H), and $B \notin$ completed(H). Then no operation in s.log(A) conflicts with an operation in s.log(B).

Proof: Straightforward by induction on the length of H. □

The next lemma uses the previous lemma to show a fundamental property of conflict-based locking: operations executed by distinct transactions that are unrelated by the precedes order do not conflict.

Lemma 13: Let H be a history in L(CBL(X)), and let A and B be transactions such that $A \neq B$, $A \notin$ aborted(H), $B \notin$ aborted(H), $(A,B) \notin$ precedes(H), and $(B,A) \notin$ precedes(H). Then no operation in OpSeq(H|A) conflicts with an operation in OpSeq(H|B).

Proof: Let K be the longest prefix of H such that neither A nor B commits in H. Since A and B are unrelated by precedes(H) and H is well-formed, OpSeq(H|A) = OpSeq(K|A), and similarly for B. Since neither A nor B is in completed(K), the result follows from Lemma 12. □

The final theorem in this section provides us with a correctness condition for the recovery algorithms to be presented in the next section: to ensure online dynamic atomicity, it is sufficient to guarantee that any subset of the active transactions can be executed in any serial order starting in the "committed state" (i.e., s.permanent), and that the final state resulting from that execution does not depend on the order.

Theorem 14: Let H be a history in L(CBL(X)). Suppose that the following holds for all prefixes K of H, where s_K is the state of CBL(X) after K:

$\forall$ commit sets C for K,
 $\forall$ total orders T,
  $\mathrm{OpSeq(Serial(K|C\text{-}committed(K),T))}(s_K.\mathrm{permanent}) \neq \perp$
  and does not depend on T.

Then H is online dynamic atomic.

Proof: The proof is by induction on the length of H. The basis case, when $H = \Lambda$, is trivial. For the inductive step, suppose that $H = K \bullet e$, where e is a single event, and assume that the theorem holds for K. Let s_K be the state of CBL(X) after K, and let s_H be the state of CBL(X) after H. If the hypothesis of the theorem is true for H, it is also true for K, since every prefix of K is a prefix of H. Thus, by the induction hypothesis, K is online dynamic atomic.

Referring to the definition of online dynamic atomicity, we need to show that, for all commit sets C for H, and for all total orders T consistent with precedes(H), $\mathrm{OpSeq(Serial(H|C,T))}(I_{S(X)})$ is defined and does not depend on T. So let C be a commit set for H, and let T1 and T2 be total orders consistent with precedes(H). Then C is a commit set for K, and T1 and T2 are consistent with precedes(K). There are now two cases.

First, if e is a commit, abort, or invocation event, notice that OpSeq throws away pending invocation events (those without corresponding response events) and completion events. Thus, OpSeq(Serial(H|C,Ti)) = OpSeq(Serial(K|C,Ti)), and the result follows from the induction hypothesis.

Second, suppose e = <ret,X,A>, where ret is a response to an invocation. If $A \notin C$, H|C = K|C, and the result follows from the induction hypothesis. So assume $A \in C$.

Let T be a total order such that the committed transactions in H are ordered by commit-order(H), A is ordered after the committed transactions, and all other transactions are ordered after A in some order. Notice that T is consistent with precedes(H). We will show that H|C is serializable in the order T, and then show that for any total order U consistent with precedes(H), OpSeq(Serial(H|C,T))($I_{S(X)}$) = OpSeq(Serial(H|C,U))($I_{S(X)}$). If we substitute first T1 and then T2 for U in this equation, the result follows from the transitivity of equality.

To show that H|C is serializable in the order T, notice that OpSeq(Serial(H|C,T)) = OpSeq(Serial(H|committed(H),T)) • OpSeq(Serial(H|C-committed(H),T)). Thus, OpSeq(Serial(H|C,T)) ($I_{S(X)}$) = OpSeq(Serial(H|C-committed(H),T)) (s_H.permanent), by Lemma 11. But by the hypotheses of the theorem, OpSeq(Serial(H|C-committed(H),T)) (s_H.permanent) is defined, so H|C is serializable in the order T.

Now let U be consistent with precedes(H). Let C1 be the subset of C ordered before A by U, and let C2 be the subset of C ordered after A by U. If we let G1 = OpSeq(Serial(H|C1,U)) and G2 = OpSeq(Serial(H|C2,U)), then OpSeq(Serial(H|C,U)) = G1 • OpSeq(H|A) • G2.

Notice that since e is a response event, precedes(H) = precedes(K) $\cup$ (committed(H)$\times$A). Therefore, committed(H) $\subseteq$ C1, so C1 is a commit set for H. Also, since $A \notin C1 \cup C2$, Gi = OpSeq(Serial(K|Ci,U)) for i=1,2.

Now, let V be a total order consistent with commit-order(H) in which the elements of committed(H) are ordered before the remaining elements of C1. Notice that V is consistent with precedes(H) (and hence precedes(K)). Let F1 = OpSeq(Serial(K|C1,V)). By the induction hypothesis, F1($I_{S(X)}$) = G1($I_{S(X)}$). Now let F2 = OpSeq(Serial(K|C1-committed(K),V)); then F1 = OpSeq(Serial(K|committed(K),V)) • F2; since K|Ci = H|Ci and committed(K) = committed(H), we can also write this as F1 = OpSeq(Serial(H|committed(H),V)) • F2, and F2 = OpSeq(Serial(H|C1-committed(H),V)).

We now show that OpSeq(Serial(H|C,U)) ($I_{S(X)}$) = OpSeq(Serial(H|C,T)) ($I_{S(X)}$). The following equations follow directly from the definitions above and the fact that F1($I_{S(X)}$) = G1($I_{S(X)}$):

OpSeq(Serial(H|C,U)) ($I_{S(X)}$)
= G1 • OpSeq(H|A) • G2 ($I_{S(X)}$)
= G2 (OpSeq(H|A) (G1 ($I_{S(X)}$)))
= G2 (OpSeq(H|A) (F1 ($I_{S(X)}$)))
= G2 (OpSeq(H|A) (OpSeq(Serial(H|committed(H),V))•F2 ($I_{S(X)}$)))
= G2 (OpSeq(H|A) (F2 (OpSeq(Serial(H|committed(H),V)) ($I_{S(X)}$))))
= G2 (OpSeq(H|A) (F2 (s_H.permanent))), by Lemma 11
= F2 • OpSeq(H|A) • G2 (s_H.permanent)

Now let W be a total order that orders the elements of C1-committed(H) before A and A before the elements of C2, and is consistent with V on C1-committed(H) and with U on C2. Then F2 • OpSeq(H|A) • G2 = OpSeq(Serial(H|C-committed(H),W)). By the hypotheses of the lemma, OpSeq(Serial(H|C-committed(H),W)) (s_H.permanent) is defined and does not depend on W. The following equations show the desired result:

F2 • OpSeq(H|A) • G2 (s_H.permanent)
= OpSeq(Serial(H|C-committed(H),W)) (s_H.permanent), as shown above
= OpSeq(Serial(H|C-committed(H),T)) (s_H.permanent),
since it does not depend on W
= OpSeq(Serial(H|C-committed(H),T))(OpSeq(Serial(H|committed(H),T))($I_{S(X)}$)),

by Lemma 11
$= \mathrm{OpSeq}(\mathrm{Serial}(H|\mathrm{committed}(H),T)) \bullet \mathrm{OpSeq}(\mathrm{Serial}(H|C\text{-}\mathrm{committed}(H),T))(I_{S(X)})$
$= \mathrm{OpSeq}(\mathrm{Serial}(H|C,T))(I_{S(X)})$

□

Let us say that an algorithm is correct if every history it permits is online dynamic atomic. The theorem shows a sufficient condition for correctness. In fact, this condition is necessary for an online algorithm. The set of histories permitted by an online algorithm is closed under the operation of taking prefixes; thus, if H is permitted by a online algorithm, so is every prefix of H. Therefore, if H is permitted by a correct online algorithm, H and each of its prefixes is online dynamic atomic. The condition stated in the theorem for each prefix K of H follows immediately from the definition of online dynamic atomicity.

6. Recovery

In this section we describe and prove correct the two recovery algorithms. The intentions list algorithm was originally presented in [28]; the undo log algorithm is presented here for the first time. The intentions list algorithm works with a conflict relation based on forward commutativity, while the undo log algorithm works with a conflict relation based on backward commutativity. Let NFC(X) ("non-forward-commuting") be the binary relation on the operations of X such that $(O_1,O_2) \in$ NFC(X) if O_1 does not commute forward with O_2 as transitions of S(X). Similarly, let NBC(X) ("non-backward-commuting") be the binary relation on the operations of X such that (O_1,O_2) $\in$ NBC(X) if O_1 does not commute backward with O_2 as transitions of S(X). The intentions list algorithm works as long as NFC(X) $\subseteq$ CONFLICT, while the undo log algorithm works as long as NBC(X) $\subseteq$ CONFLICT. At the end of the section we present some examples that illustrate that the two other combinations of recovery algorithms with conflict relations do not work correctly.

6.1. The Intentions Lists Algorithm

Using intentions lists for recovery, when a transaction executes an operation (i.e., a return value is generated for the operation), the operation is simply recorded in the log for the transaction; the permanent state is not modified until the transaction commits. In addition, to choose a return value for the operation, the operation is executed in a state derived from the permanent state by applying the transaction's prior operations (i.e., the operations already in its log). Thus, as discussed earlier, an operation does not see the effects of operations executed by other active transactions.

The intentions list algorithm is described by a state machine ILIST(X). ILIST(X) has the same transitions as CBL(X). A state s of ILIST(X) has the same five components as a state of CBL(X), and the initial state is the same as for CBL(X). The transition function for ILIST(X) is the same as that for CBL(X), except for response events, which have the additional precondition shown below. (For brevity, and to highlight the differences between the two algorithms, we list only the modifications to the description of CBL(X).)

<ret, X, A>
  Precondition:
    new-log(s'.permanent) $\neq \perp$

Using intentions lists, a transaction's view is the committed state, modified by its log. Thus, the possible responses to an invocation are those that are permitted in the transaction's view. This constraint is embodied in the precondition above.

We now show that the intentions list algorithm is correct as long as NFC(X) $\subseteq$ CONFLICT.

We begin with two lemmas describing the relationship between ILIST(X) and CBL(X); their proofs are straightforward. The first lemma shows that every history in L(ILIST(X)) is also in L(CBL(X)); in other words, ILIST(X) simply restricts the behavior of CBL(X).

Lemma 15: L(ILIST(X)) $\subseteq$ L(CBL(X)).

The second lemma shows that the state of ILIST(X) after a history H is the same as the state of CBL(X) after the same history.

Lemma 16: Let H be a history in L(ILIST(X)), let s_{IL} be the state of ILIST(X) after H, and let s_{CBL} be the state of CBL(X) after H. Then $s_{IL} = s_{CBL}$.

The two lemmas above allow us to apply the results of the previous section about CBL(X) to ILIST(X); in the remainder of this section we will do so implicitly, without referring directly to the two lemmas above.

The next lemma shows that an active transaction's log is defined in the committed state. This is clearly true as long as no transaction commits, given the precondition on response events. The key to the lemma is that when a transaction commits, none of its operations conflict with the operations executed by other active transactions; thus, the operations executed by the other active transactions will still be defined in the new committed state.

Lemma 17: Suppose NFC(X) $\subseteq$ CONFLICT. Let H be a history in L(ILIST(X)), let s be the state of ILIST(X) after H, and let A be a transaction such that A $\notin$ completed(H). Then s.log(A) is defined in s.permanent.

Proof: The proof is by induction on the length of H. The basis, when H = Λ, is trivial. For the induction step, assume that H = K•e, where e is a single event, and that the lemma holds for K. Let s_K be the state of ILIST(X) after K. There are 3 cases, depending on the type of e:

1. If e is an invocation or abort event, s.permanent = s_K.permanent, and s.log = s_K.log. Thus, the result follows from the induction hypothesis.
2. If e = <ret,X,B> where ret is a response to an invocation, then s.permanent = s_K.permanent. If B≠A, s.log(A) = s_K.log(A), and the result follows from the induction hypothesis. Otherwise, the precondition for e guarantees the desired result.
3. If e = <commit,X,B>, then s.log = s_K.log. If B$\in$ committed(K) then s.permanent = s_K.permanent, and the result follows from the induction hypothesis. Otherwise, s.permanent = s.log(B)(s_K.permanent). Since B$\notin$ committed(K) and A$\notin$ committed(H), (B,A)$\notin$ precedes(H) and (A,B)$\notin$ precedes(H). Therefore, by Lemma 13, no operation in s.log(A) conflicts with an operation in s.log(B). Since NFC(X) $\subseteq$ CONFLICT, every operation in s.log(A) commutes forward with every operation in s.log(B). Therefore, by Corollary 5, s.log(A) commutes forward with s.log(B). By the induction hypothesis, s.log(A) and s.log(B) are both defined in s_K.permanent. Thus, by the definition of forward commutativity, s.log(A) is defined in s.log(B)(s_K.permanent), which as argued above is simply s.permanent.

□

We now show that the intentions list algorithm meets the criteria established in Theorem 14, as long as NFC(X) $\subseteq$ CONFLICT.

Lemma 18: Suppose NFC(X) $\subseteq$ CONFLICT. Let H be a history in L(ILIST(X)), let s be the state of ILIST(X) after H, and let C be a commit set for H. Then for all total orders T on transactions, OpSeq(Serial(H|C-committed(H),T))(s.permanent) $\neq \perp$ and does not depend on T.

Proof: The result follows directly from Lemmas 17, 12, and 6. □

The final theorem demonstrates the correctness of the intentions list algorithm.

Theorem 19: Suppose NFC(X) $\subseteq$ CONFLICT. Let H be a history in L(ILIST(X)). Then H is online dynamic atomic.

Proof: The result follows directly from Lemma 18 and Theorem 14. □

6.2. The Undo Log Algorithm

The undo log algorithm keeps track of the "current" state of the object, which is the state obtained by executing the committed *and active* transactions starting in the initial state. When a transaction executes an operation (i.e., a return value is generated for the operation), the operation is recorded in the log for the transaction, and the current state is modified to reflect the effects of the operation. A return value for an operation is chosen by executing the operation in the *current* state, so an operation's view includes the effects of other active transactions.

We describe the handling of recovery in a very general way. Rather than postulating the existence of "undo operations" for each ordinary operation executed by a transaction, we describe the effect that such undo operations must achieve. We will explain how recovery is handled in more detail after we present the algorithm.

The undo log algorithm is described by a state machine ULOG(X). ULOG(X) has the same transitions as CBL(X). A state s of ULOG(X) has the same five components as a state of CBL(X), and in addition a component s.current, which is a state of S(X). The initial states of the first five components are the same as for CBL(X), and the initial state of the last component is $I_{S(X)}$. The transition function for ULOG(X) is the same as that for CBL(X), except for response and abort events, which have the additional preconditions and postconditions shown below. (As for ILIST(X), we list only the modifications to the description of CBL(X).)

```
<abort, X, A>
  Postcondition:
      s.current = active(s'.permanent),
              where active is any concatenation of the sequences in the set
              $\{s'.log(B) \mid B \notin s'.committed \cup s'.aborted \cup \{A\}\}$

<ret, X, A>
  Precondition:
      new-op(s'.current) $\neq \perp$
    Postcondition:
      s.current = new-op(s'.current)
```

The current state reflects the operations executed by both committed and active transactions. The responses permitted for an invocation are those that are permitted in the current state, as indicated by the precondition for response events. The postcondition for a response event indicates that the current state is modified to reflect the effects of the operation just executed when a response is returned. Aborts are handled by finding a new current state that bears the appropriate relationship to the permanent state.

We should point out that a real implementation would probably maintain only the current state, along with a log that allows it to undo operations in the event of an abort. Thus, the permanent state need not be stored explicitly (although it must be computable from stored information, since all active transactions could abort). The presence of the permanent state as a component of the state of ULOG(X) allows us to state the effect that must be achieved by an abort, without indicating how an implementation must achieve it. For example, we do not need to postulate the existence of undo operations corresponding to each ordinary operation.

We now show that the undo log algorithm is correct as long as NBC(X) $\subseteq$ CONFLICT.

As in our proof of ILIST(X), we begin with two lemmas describing the relationship between ULOG(X) and CBL(X); their proofs are straightforward. The first lemma shows that every history in L(ULOG(X)) is also in L(CBL(X)); in other words, ULOG(X) simply restricts the behavior of CBL(X).

Lemma 20: L(ULOG(X)) $\subseteq$ L(CBL(X)).

The second lemma shows the relationship between the state of ULOG(X) after a history H and the state of CBL(X) after the same history.

Lemma 21: Let H be a history in L(ULOG(X)), let s_{UL} be the state of ULOG(X) after H, and let s_{CBL} be the state of CBL(X) after H. *Then* s_{UL}.*permanent* = s_{CBL}.permanent, s_{UL}.pending = s_{CBL}.pending, s_{UL}.log = s_{CBL}.log, s_{UL}.committed = s_{CBL}.committed, and s_{UL}.aborted = s_{CBL}.aborted.

The two lemmas above allow us to apply the results of the previous section about CBL(X) to ULOG(X); in the remainder of this section we will do so implicitly, without referring directly to the two lemmas above.

The next lemma describes an important invariant about the "permanent" and "current" components of the state of ULOG(X).

Lemma 22: Suppose NBC(X) $\subseteq$ CONFLICT. Let H be a history in L(ULOG(X)), let s be the state of ULOG(X) after H, and let Active = ACT-committed(H)-aborted(H). Then for all total orders T consistent with precedes(H):

1. s.permanent = OpSeq(Serial(H|committed(H),T))($I_{S(X)}$) $\neq \perp$
2. s.current = OpSeq(Serial(H|Active,T))(s.permanent) $\neq \perp$

Proof: The proof is by induction on the length of H. The basis, when H = Λ, is trivial. For the induction step, let H = K•e, where e is a single event, and assume that the lemma holds for K. Let s_K be the state of ULOG(X) after K.

Suppose that T and U are consistent with precedes(H). Since NBC(X) $\subseteq$ CONFLICT, it follows from Lemmas 9 and 13 that OpSeq(Serial(H|committed(H),T))($I_{S(X)}$) = OpSeq(Serial(H|committed(H),U))($I_{S(X)}$), and that OpSeq(Serial(H|Active,T))(s.permanent) = OpSeq(Serial(H|Active,U))(s.permanent). Thus, it suffices to find a single total order T consistent with precedes(H) such that the two conditions in the statement of the lemma hold for T. There are now four cases, depending on the type of e:

1. If e = <inv,X,A>, where inv is an invocation, and T is any total order on transactions, then s.permanent = s_K.permanent, s.current = s_K.current, OpSeq(Serial(H|committed(H),T)) = OpSeq(Serial(K|committed(K),T)), and OpSeq(Serial(H|Active,T)) = OpSeq(Serial(K|Active,T)). Thus, the result follows from the induction hypothesis.

2. Suppose e = <commit,X,A>. If A$\in$committed(K), then s.permanent = s_K.permanent, s.current = s_K.current, OpSeq(Serial(H|committed(H),T)) = OpSeq(Serial(K|committed(K),T)), and OpSeq(Serial(H|Active,T)) = OpSeq(Serial(K|Active,T)). Thus, the result follows from the induction hypothesis. Otherwise, let T be a total order consistent with commit-order(H) that orders the elements of committed(H) before all other transactions.

   By Lemma 11, s.permanent = OpSeq(Serial(H|committed(H),T))($I_{S(X)}$). Thus, for the first part of the lemma, it suffices to show that s.permanent $\neq \perp$. By the induction hypothesis, s_K.current = OpSeq(Serial(K|Active$\cup${A},T))(s_K.permanent) $\neq \perp$. By construction, T orders A before the elements of Active. Therefore, OpSeq(Serial(K|Active$\cup${A},T)) = OpSeq(K|A)•OpSeq(Serial(K|Active,T)). But OpSeq(K|A) = OpSeq(H|A). Therefore, OpSeq(H|A) is a prefix of a sequence defined in s_K.permanent, and hence is itself defined in s_K.permanent. By the postconditions for e, s.permanent = OpSeq(H|A)(s_K.permanent). Since OpSeq(H|A) is defined in s_K.permanent, s.permanent $\neq \perp$.

   Now we show that s.current = OpSeq(Serial(H|Active,T))(s.permanent) $\neq \perp$. Notice that s.current = s_K.current, and s.permanent = s_K.log(A)(s_K.permanent). By the induction hypothesis, s_K.current $\neq \perp$. Therefore, it suffices to show that s_K.current = OpSeq(Serial(H|Active,T))(s.permanent). By the induction hypothesis, s_K.current = OpSeq(Serial(K|Active$\cup${A},T))(s_K.permanent). Since T orders A before the elements of Active,

$$\begin{aligned} s_K.\text{current} &= \text{OpSeq(Serial(K|Active}\cup\{A\},T))(s_K.\text{permanent}) \\ &= \text{OpSeq(K|A)} \bullet \text{OpSeq(Serial(K|Active,T))}(s_K.\text{permanent}) \\ &= \text{OpSeq(Serial(K|Active,T))}\ (\ \text{OpSeq(K|A)}\ (s_K.\text{permanent})\ ) \\ &= \text{OpSeq(Serial(H|Active,T))}\ (\ \text{OpSeq(K|A)}\ (s_K.\text{permanent})\ ) \\ &= \text{OpSeq(Serial(H|Active,T))}\ (\ s_K.\text{log(A)}\ (s_K.\text{permanent})\ ) \end{aligned}$$

= OpSeq(Serial(H|Active,T)) (s.permanent)

This completes the argument for e = <commit,X,A>.

3. Suppose e = <ret,X,A>, where ret is a response to an invocation. Let op = X:[s_K.pending(A),ret]. First, note that s.permanent = s_K.permanent. Furthermore, committed(H) = committed(K), and H|committed(H) = K|committed(K). Thus, the first half of the lemma follows from the induction hypothesis.

   Now, by the preconditions for e, op(s_K.current) $\neq \perp$. By the postconditions, s.current = op(s_K.current). Thus, s.current $\neq \perp$. Now let T be a total order that orders A after all other elements of Active. The following equations show that OpSeq(Serial(H|Active,T))(s.permanent) = s.current:

   OpSeq(Serial(H|Active,T)) (s.permanent)
   = OpSeq(Serial(H|Active-{A},T))•OpSeq(H|A) (s.permanent)
   = OpSeq(H|A) (OpSeq(Serial(H|Active-{A},T)) (s.permanent))
   = OpSeq(K|A)•op (OpSeq(Serial(K|Active-{A},T)) (s.permanent))
   = op (OpSeq(K|A) (OpSeq(Serial(K|Active-{A},T)) (s.permanent)))
   = op (OpSeq(Serial(K|Active-{A},T))•OpSeq(K|A) (s.permanent))
   = op (OpSeq(Serial(K|Active,T)) (s.permanent))
   = op (s_K.current)
   = s.current

4. Finally, suppose e = <abort,X,A>. Then s.permanent = s_K.permanent, committed(H) = committed(K), and H|committed(H) = K|committed(K). Thus, the first half of the lemma follows from the induction hypothesis.

   Now, by the postconditions for e, there exists a total order T such that s.current = OpSeq(Serial(H|Active,T))(s_K.permanent). Since s.permanent = s_K.permanent, it suffices to show that OpSeq(Serial(H|Active,T))(s_K.permanent) $\neq \perp$. Let U be a total order consistent with T on Active that orders A after the elements of Active. By the induction hypothesis, s_K.current $\neq \perp$. But

   s_K.current = OpSeq(Serial(K|Active$\cup${A},U))(s_K.permanent),
   by the induction hypothesis
   = OpSeq(Serial(K|Active,U))•OpSeq(K|A) (s_K.permanent)
   = OpSeq(Serial(H|Active,T))•OpSeq(H|A) (s.permanent)
   = OpSeq(H|A) (OpSeq(Serial(H|Active,T)) (s.permanent))

   Therefore, OpSeq(Serial(H|Active,T))(s_K.permanent) $\neq \perp$.

□

The final lemma shows that ULOG(X) meets the criteria established in Theorem 14.

Lemma 23: Suppose NBC(X) $\subseteq$ CONFLICT. Let H be a well-formed sequence in L(ULOG(X)), let s be the state of ULOG(X) after H, and let C be a commit set for H. Then for all total orders T on transactions, OpSeq(Serial(H|C-committed(H),T))(s.permanent) $\neq \perp$ and does not depend on T.

Proof: Since NBC(X) $\subseteq$ CONFLICT, it follows from Lemmas 9 and 13 that OpSeq(Serial(H|C-committed(H),T))(s.permanent) does not depend on T. To show that OpSeq(Serial(H|C-committed(H),T))(s.permanent) $\neq \perp$, let Active = ACT-committed(H)-aborted(H), and let U be a total order consistent with precedes(H) that is also consistent with T on C-committed(H), such that U orders the elements of C-committed(H) before the other elements of Active. By Lemma 22, OpSeq(Serial(H|Active,U))(s.permanent) $\neq \perp$. By choice of U, OpSeq(Serial(H|Active,U)) = OpSeq(Serial(H|C-committed(H),T)) • OpSeq(Serial(H|Active-C,U)). Thus, OpSeq(Serial(H|C-committed(H),T)) is a prefix of a sequence that is defined in s.permanent, and hence is itself defined in s.permanent. □

Finally, we show that ULOG(X) is correct.

Theorem 24: Suppose NBC(X) $\subseteq$ CONFLICT. Let H be a well-formed sequence in L(ULOG(X)).

Then H is online dynamic atomic.

Proof: The result follows directly from Lemma 23 and Theorem 14. □

6.3. Remarks

The theorems above show that the intentions list algorithm works as long as the CONFLICT relation includes all non-forward-commuting pairs of operations, and that the undo log algorithm works as long as the CONFLICT relation includes all non-backward-commuting pairs of operations. One might ask whether the other combinations of recovery algorithms and conflict relations are guaranteed to work. The answer, unfortunately, is no.

For example, consider the bank account object Y defined earlier. Successful withdrawal operations commute backward, but not forward. Suppose we use a conflict relation equal to NBC(Y) in ILIST(Y). Suppose one transaction deposits, say, \$3, and then commits, so the permanent state is \$3, and then two transactions concurrently each withdraw \$3. Each operation is defined in the appropriate transaction's view (which is just the permanent state). If the two transactions then commit, however, the resulting execution is not dynamic atomic; in fact, it is not even atomic, since there is no order in which the committed transactions can be serialized without violating the serial specification of Y.

Similarly, notice that successful withdrawals commute forward with deposits, but not backward. A scenario similar to that above demonstrates that a conflict relation equal to NFC(Y) does not work in ULOG(Y).

7. Discussion

We have presented two novel concurrency control algorithms for abstract data types that use conflict relations derived from the specifications of the types to permit high levels of concurrency. The two algorithms use different recovery methods, and consequently require subtly different conflict relations. The algorithms generalize prior work by permitting operations to be both partial and non-deterministic, and by allowing information about the results of an operation to be used in determining the lock required by the operation. We have proved that the two algorithms ensure *online dynamic atomicity*, thus demonstrating not just that they ensure atomicity, but that they do so in any system in which each object uses a dynamic atomic concurrency control algorithm. Thus, the choice of a particular concurrency control algorithm can be made locally for each object in a system, rather than requiring the entire system to use the same algorithm. This flexibility is particularly important in loosely coupled distributed systems.

One area for further work involves nested transactions [18, 23, 14]. Working with Alan Fekete, Nancy Lynch, and Michael Merritt, we have generalized the definitions of dynamic atomicity in this paper to a system involving nested transactions. We have also generalized the intentions list algorithm described in this paper to handle nested transactions, and have proved it correct. We expect to be able to generalize the undo log algorithm in a similar way.

While we have shown that the intentions list algorithm works with a conflict relation based on forward commutativity, and the undo log algorithm with one based on backward commutativity, and furthermore that the other combinations are not correct, we do not know whether the two notions of commutativity are in any sense "optimal" for their respective recovery algorithms. However, it should be possible to generalize the examples in Section 6.3 to obtain some sort of optimality results.

It is sometimes convenient to combine different recovery methods in the same system. For example, some operations on an object might be handled using intentions lists, while others might be handled using undo logs. The algorithms we have presented in this paper allow different recovery methods to be used in distinct objects, but not in the same object. We are currently working on an algorithm that permits the two methods to be combined by using a slightly more complicated concurrency control technique.

In this paper we have discussed only recovery from aborts of transactions; the model we used does not cover crashes. Crash recovery is similar to, but typically more complex than, recovery from aborts. Further work is needed to study the applicability of our algorithms to crash recovery, and to analyze the interactions of crash recovery with abort recovery and concurrency control.

While other work on commutativity-based concurrency control treats recovery separately or ignores it altogether, most of it seems to be assuming that recovery is performed using the undo log algorithm above, in the sense that operations are executed in the *current* state. The algorithms in this paper demonstrate that the choice of recovery algorithm has a subtle impact on the concurrency control algorithm.

We have identified a fundamental difference in the two recovery algorithms, namely the "view" used to choose the results of an operation. We have also presented a correctness condition that concurrency control and recovery must ensure together to satisfy online dynamic atomicity. We are currently studying whether this condition applies to other algorithms.

References

[1] Allchin, J. E.
An architecture for reliable decentralized systems.
PhD thesis, Georgia Institute of Technology, September, 1983.
Available as Technical Report GIT-ICS-83/23.

[2] Beeri, C., et al.
A concurrency control theory for nested transactions.
In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 45-62. ACM, Montreal, Canada, August, 1983.

[3] Bernstein, P. A., and Goodman, N.
Concurrency control in distributed database systems.
ACM Computing Surveys 13(2):185-221, June, 1981.

[4] Bernstein, P., Goodman, N., and Lai, M.-Y.
Two part proof schema for database concurrency control.
In *Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 71-84. February, 1981.

[5] Bernstein, P., V. Hadzilacos, N. Goodman.
Concurrency control and recovery in database systems.
Addison-Wesley, 1987.

[6] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L.
The notions of consistency and predicate locks in a database system.
Communications of the ACM 19(11):624-633, November, 1976.

[7] Gray, J.
Notes on Database Operating Systems.
In *Lecture Notes in Computer Science.* Volume 60: *Operating Systems -- An Advanced Course.* Springer-Verlag, 1978.

[8] Gray, J.N., et al.
The recovery manager of the System R database manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[9] Hadzilacos, V.
A theory of reliability in database systems.
Journal of the ACM 35(1):121-145, January, 1988.

[10] Herlihy, M, N. Lynch, M. Merritt, W. Weihl.
On the correctness of orphan elimination algorithms (preliminary report).
In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing.* IEEE, Pittsburgh, July, 1987.

[11] Korth, H. F.
Locking Primitives in a Database System.
Journal of the Association for Computing Machinery 30(1):55-79, January, 1983.

[12] Kung, H.T., and Robinson, J.T.
On optimistic methods for concurrency control.
ACM Transactions on Database Systems 6(2):213-226, June, 1981.

[13] Lampson, B.
Atomic transactions.
In Goos and Hartmanis (editors), *Lecture Notes in Computer Science.* Volume 105: *Distributed Systems: Architecture and Implementation*, pages 246-265. Springer-Verlag, Berlin, 1981.

[14] Liskov, B., and Scheifler, R.
Guardians and actions: linguistic support for robust, distributed programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[15] Liskov, B., Scheifler, R., Walker, E. F., and Weihl, W.
Orphan Detection (Extended Abstract).
In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*. IEEE, July, 1987.

[16] Lynch, N. A., and M. R. Tuttle.
Hierarchical correctness proofs for distributed algorithms.
Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, April, 1987.

[17] Lynch, N. A., and Merritt, M.
Introduction to the theory of nested transactions.
Technical Report MIT-LCS-TR-367, Massachusetts Institute of Technology, 1986.
Appeared in Proceedings of 1986 International Conference on Database Theory.

[18] Moss, J.E.B.
Nested transactions: an approach to reliable distributed computing.
PhD thesis, Massachusetts Institute of Technology, 1981.
Available as Technical Report MIT/LCS/TR-260.

[19] Moss, J., N. Griffeth, M. Graham.
Abstraction in concurrency control and recovery management (revised).
Technical Report COINS Technical Report 86-20, University of Massachusetts at Amherst, May, 1986.

[20] Nelson, B. J.
Remote procedure call.
PhD thesis, Carnegie-Mellon University Department of Computer Science, May, 1981.
Available as CMU-CS-81-119.

[21] O'Neil, P. E.
The escrow transactional method.
ACM Transactions on Database Systems 11(4):405-430, December, 1986.

[22] Papadimitriou, C.H.
The serializability of concurrent database updates.
Journal of the ACM 26(4):631-653, October, 1979.

[23] Reed, D.P.
Naming and synchronization in a decentralized computer system.
PhD thesis, Massachusetts Institute of Technology, 1978.
Available as Technical Report MIT/LCS/TR-205.

[24] Schwarz, P.
Transactions on typed objects.
PhD thesis, CMU, December, 1984.
Available as Technical Report CMU-CS-84-166.

[25] Schwarz, P. M., and Spector, A. Z.
Synchronizing shared abstract types.
ACM Transactions on Computer Systems 2(3):223-250, August, 1984.

[26] Skeen, M. D.
Crash recovery in a distributed database system.
PhD thesis, University of California at Berkeley, May, 1982.
Available as UCB/ERL M82/45.

[27] Spector, A. Z., et al.
Support for distributed transactions in the TABS prototype.
IEEE Transactions on Software Engineering SE-11(6):520-530, June, 1985.

[28] Weihl, W. E.
Specification and implementation of atomic data types.
PhD thesis, Massachusetts Institute of Technology, 1984.
Available as Technical Report MIT/LCS/TR-314.

[29] Weihl, W. E.
Local atomicity properties: modular concurrency control for abstract data types.
ACM Transactions on Programming Languages and Systems , 1987.
Accepted for publication.

[30] Weihl, W., and Liskov, B..
Implementation of resilient, atomic data types.
ACM Transactions on Programming Languages and Systems , April, 1985.

[31] Weikum, G.
A theoretical foundation of multi-level concurrency control.
In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, pages 31-42. 1986.

[32] Weikum, G., and H.-J. Schek.
Architectural issues of transaction management in multi-layered systems.
In *Proceedings of the Tenth International Conference on Very Large Data Bases*, pages 454-465. Singapore, August, 1984.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy