4

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-368

HYBRID CONCURRENCY CONTROL FOR ABSTRACT DATA TYPES

Maurice P. Herlihy
William E. Weihl

August 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

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-368	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)*
Hybrid Concurrency Control for Abstract Data Types

12. PERSONAL AUTHOR(S)
Herlihy, Maurice P. and 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	26

16. SUPPLEMENTARY NOTATION

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, time-stamps, local atomicity, hybrid atomicity, abstract data types

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

We define a new locking protocol that permits more concurrency than existing commutativity-based protocols. In addition, the protocol permits operations to be both partial and non-deterministic, and it permits the lock mode for an operation to be determined by its results as well as its name and arguments. The protocol exploits type-specific properties of objects; necessary and sufficient constraints on lock conflicts are defined directly from a data type specification. We give a complete formal description of the protocol, encompassing both concurrency control and recovery, and prove that the protocol satisfies hybrid atomicity, a local atomicity property that combines aspects of static and dynamic atomic protocols. We also show that the protocol is optimal in the sense that no hybrid atomic locking scheme can permit more concurrency.

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)* (517) 253-5894
22c. OFFICE SYMBOL	

Hybrid Concurrency Control for Abstract Data Types

Maurice P. Herlihy[1]

William E. Weihl[2]

29 July 1988

Abstract

We define a new locking protocol that permits more concurrency than existing commutativity-based protocols. In addition, the protocol permits operations to be both partial and non-deterministic, and it permits the lock mode for an operation to be determined by its results as well as its name and arguments. The protocol exploits type-specific properties of objects; necessary and sufficient constraints on lock conflicts are defined directly from a data type specification. We give a complete formal description of the protocol, encompassing both concurrency control and recovery, and prove that the protocol satisfies *hybrid atomicity*, a local atomicity property that combines aspects of static and dynamic atomic protocols. We also show that the protocol is optimal in the sense that no hybrid atomic locking scheme can permit more concurrency.

M. Herlihy is supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. W. Weihl is supported in part by the National Science Foundation under Grants DCR-8510014 and CCR-8716884, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

[1]Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. (Herlihy@CS.CMU.EDU)

[2]MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. (Weihl@XX.LCS.MIT.EDU)

1. Introduction

Atomic transactions are a widely accepted mechanism for coping with failures and concurrency in database systems, both distributed and centralized. Many algorithms have been proposed for concurrency control and recovery [1]. Early work in this area considered only untyped objects: operations were either left uninterpreted, or were treated simply as reads or writes. More recent work has focused on typed objects, such as queues, directories, or counters, that provide a richer set of operations. Several algorithms have been proposed to enhance concurrency and recovery by exploiting data objects' type-specific properties [2, 13, 18, 22]. Most of these algorithms are locking schemes in which conflicts are governed by some notion of commutativity: lock modes for commuting operations do not conflict.

This paper presents a new locking algorithm for concurrency control and recovery of typed data objects. As discussed below, our algorithm permits more concurrency than many type-specific locking schemes in the literature [2, 5, 13, 18, 22]: our algorithm places fewer constraints on lock conflicts, thus permitting a larger set of interleavings. Moreover, our algorithm is "upwardly compatible" with these other schemes in the sense that they can be used together in the same system without jeapordizing serializability or recovery.

In most of the type-specific algorithms in the literature, lock conflicts are governed by some notion of commutativity: if two operations commute, their locks need not conflict. Informally, this condition arises in conventional two-phase locking schemes as follows. If two transactions attempt to acquire conflicting locks, one must wait for the other to complete. The induced delay ensures that the latter is serialized before the former. Two-phase locking thus determines transaction serialization up to a partial order: transactions unrelated by the transitive closure of this lock conflict relation may be serialized in an arbitrary order. Moreover, such unrelated transactions may be serialized in different orders at different data objects, or at different sites in a distributed system. If the operations of concurrent transactions commute, then all such local orderings are equivalent and compatible with a global total serialization ordering.

The basic idea behind our algorithm is quite simple. Transactions are serializable in the order they commit. As part of each transaction's commitment protocol, it generates a timestamp from a logical clock, and distributes that timestamp to the objects it updated.[3] Our algorithm augments the implicit partial order induced by lock conflicts with the explicit total order induced by transactions' commit timestamps. By making the serialization order explicit, we can replace the commutativity requirement with a weaker notion, which we call dependency. For example, our algorithms permits concurrent transactions to enqueue on a FIFO queue, even though the enqueue operations do not commute.

Our algorithm is quite general: it works for arbitrary data types, including types with partial and non-deterministic operations. Our treatment is systematic: necessary and sufficient conditions for locks to conflict are derived by analyzing the object's data type specification. We give a formal characterization of our notion of conflict, and we prove our algorithm is correct. Because concurrency control and recovery interact in subtle ways, our descriptions and proofs encompass both concurrency and recovery.

Section 2 defines our model of computation, and Section 3 gives a formal definition of atomicity. Section 4 describes our criteria for lock conflict, and Section 5 describes our algorithm and proves it correct. Section 6 discusses some pragmatic issues. Finally, section 7 closes with a discussion and summary.

[3]These commit timestamps should not be confused with the timestamps used in multiversion protocols such as Reed's [17], in which transactions are serialized in a statically predefined order.

2. Model of Computation

Our model of computation [22, 23] has two kinds of entities: *transactions* and *objects*. Each object provides operations that can be called by transactions to examine and modify the object's state. These operations constitute the sole means by which transactions can access the state of the object. We typically use the symbols P, Q, and R for transactions, and X, Y, and Z for objects.

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, denoted <inv, X, P>, occur when a transaction P invokes an operation of object X. The "inv" field includes both the name of the operation and its arguments.
- Response events, denoted <res, X, P>, occur when an object returns a response res to an earlier invocation by transaction P of an operation of object X.
- Commit events, denoted <*commit(t)*, X, P>, occur when object X learns that transaction P has committed with timestamp t. Timestamps are taken from a countable, totally ordered set.
- Abort events, denoted <abort, X, P>, occur when object X learns that transaction P has aborted.

We refer to commit and abort events collectively as *completion* events. We say that event <e, X, P> *involves* X and P.

We introduce some notation here. The symbol "•" denotes concatenation of sequences, and the symbol "Λ" denotes the empty sequence. 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 $\mathcal{P}$ is a set of transactions, we define $H|\mathcal{P}$ similarly. If X is an object and P is a transaction, we write H|X for H|{X}, and H|P for H|{P}. We define *committed(H)* to be the set of transactions for which commit events occur in H, and *aborted(H)* to be the set of transactions for which abort events occur. We also define *completed(H)* to be committed(H) $\cup$ aborted(H), the set of transactions that commit or abort in H.

Not all sequences of events make sense as computations. For example, a transaction should not commit at some objects and abort at others, or commit with different timestamps at different objects. To capture these constraints, we introduce a set of *well-formedness* constraints. A well-formed sequence of events is called a *history*. We divide our well-formedness constraints into two parts: constraints on the execution of individual transactions, and constraints on the timestamps that can appear in commit events. Individual transactions are constrained as follows:

- Each transaction P must wait for the response to its last invocation before invoking the next operation, and an object can generate a response for P only if P has a pending invocation. More precisely, let *op-events(H|P)* be the subsequence of H|P consisting of all invocation and response events; op-events(H|P) 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 event must involve the same object.
- Each transaction P can commit or abort in H, but not both; i.e., committed(H|P) $\cap$ aborted(H|P) = $\emptyset$.
- A transaction P cannot commit if it is waiting for the response to an invocation, and cannot invoke any operations after it commits. More precisely, if P $\in$ committed(H|P), then H|P consists of op-events(H|P) followed by some number of commit events, and op-events(H|P) ends in a response event.

These restrictions on transactions are intended to model the typical use of transactions in existing systems. 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. A transaction is not allowed to commit at some objects and abort at others; this requirement, called *atomic commitment*, can be implemented using well-known commitment protocols [7, 15, 19].

There are two additional constraints, which simply state that the timestamps chosen for transactions are unique, and that a transaction chooses only one timestamp.

- Any two commit events in H for the same transaction have the same timestamp.
- Any two commit events in H for different transactions have different timestamps.

We place 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 [6, 16], and we would like our results to be as generally applicable as possible.

3. Atomicity

In this section we define atomicity and several related properties. The definitions are abstracted from [22, 23]. Unlike many earlier models that classify operations only as reads or writes, our model emphasizes abstraction, in particular data abstraction. Atomicity is defined in terms of objects' specifications, so that transactions are atomic if their execution appears to be serializable and recoverable to transactions, given only the specifications of the objects. For example, a system may be atomic at one level of abstraction and non-atomic at lower levels.

3.1. Specifications

Each object has a *serial specification*, which defines its behavior in the absence of concurrency and failures. An object's serial specification is a set of *operation sequences*. An *operation* is a pair consisting of an invocation and a matching response. In addition, an operation identifies the object on which it is executed. We often speak informally of an "operation" on an object, as in "the enq operation on a queue 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 queue object X:

X:[Enq(3),Ok]

This operation represents an execution of the Enq operation of X with argument "3" and result "Ok." For brevity, we often say that an operation sequence is *legal* if it belongs to the serial specification currently of interest.

Each object also has a *behavioral specification*, which characterizes its behavior in the presence of concurrency and failures. An object's behavioral specification is just a set of histories that contain events involving that object only.

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 order to exploit type-specific properties, we need to define serializability and atomicity in terms of the serial specifications of objects.

Since serial specifications are sets of operation sequences, not sets of histories, we need to establish a correspondence between histories and operation sequences. We say that a history is *serial* if events for different transactions are not interleaved. If H is a serial history, and $P_1, ..., P_n$ are the transactions in H in the order in which they appear, then we can write H as $H|P_1 \bullet ... \bullet H|P_n$. 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 P_i, OpSeq($H|P_i$) is the operation sequence obtained from $H|P_i$ by pairing each invocation event with its corresponding termination event, and discarding commit events and pending invocation events. For the full history H, OpSeq(H) is defined to be OpSeq($H|P_1$)•...•OpSeq($H|P_n$).

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

<Enq(3),X,Q>
<Ok,X,Q>
<Commit(t1),X,Q>
<Deq(),X,P>
<3,X,P>
<commit(t2),X,P>

then OpSeq(H) is the operation sequence

X:[Enq(3),Ok]
X:[Deq(),3]

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.

Two histories H and K are *equivalent* if every transaction performs the same sequence of steps in each, i.e., if H|P = K|P for every transaction P. 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 P_1, ..., P_n are the transactions in H in the order T, then Serial(H,T) = $H|P_1 \bullet \ldots \bullet H|P_n$.

Let T be a total ordering of transactions. 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.

For example, the following history involving a first-in-first-out (FIFO) queue X is atomic:

<Enq(1), X, P>
<Ok, X, P>
<Enq(2), X, Q>
<Ok, X, Q>
<Enq(3), X, P>
<Ok, X, P>
<commit(2), X, P>
<commit(1), X, Q>
<Deq, X, R>
<2, X, R>
<Deq, X, R>
<1, X, R>
<commit(5), X, R>

The history contains only committed transactions, and is serializable in the order Q followed by P followed by R.

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. 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. To design a local atomicity property, one must ensure that the objects agree on at least one serialization order for the committed transactions. This problem can be difficult because each object has only *local* information; no object has complete information about the *global* computation of the system. As illustrated in [22, 23], 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.

In this section we define a particular local atomicity property, which we call *hybrid atomicity*. This local atomicity property uses the timestamps chosen when transactions commit to constrain each object's local serialization order. The only difficulty is that an object does not know what timestamp will be chosen by a transaction until the transaction commits. This difficulty is alleviated by placing certain simple constraints on the timestamp generation method. If H is a history, define *precedes*(H) to be the following relation on transactions: (P,Q) $\in$ precedes(H) if and only if there exists an operation invoked by Q that returns a result after P commits in H. The relation precedes(H) captures potential "information flow" between transactions: if (P,Q) $\in$ precedes(H), then some operation executed by Q occurred in H after P committed, hence Q may have acquired a lock released by P, which would imply that Q must be serializable after P.

Now, let *TS*(H) be the partial order on transactions defined by (P,Q) $\in$ TS(H) if P and Q commit in H and the timestamp for P is less than the timestamp for Q. We require the timestamp generation method to satisfy the following constraint: the timestamp order on committed transactions must be consistent with the precedes order at each object. In other words, precedes(H|X) $\subseteq$ TS(H) for all objects X. Informally, this constraint requires that if Q runs at X and sees that P has already committed, then Q must choose a timestamp greater than P's. This constraint is satisfied by timestamp generation algorithms based on logical clocks [14], and by algorithms that piggyback timestamp information on the messages of a commit protocol.

A history H is *hybrid atomic* if permanent(H) is serializable in the order TS(H). (Notice that TS(H) defines a total order on committed(H).) An object is hybrid atomic if every history permitted by its behavioral specification is hybrid atomic. Hybrid atomicity is a local atomicity property [22, 23]:

> **Theorem 1:** If every object in a system is hybrid atomic, then every history in the system's behavior is atomic.

As an aside, we remark that hybrid atomicity is an *optimal* local atomicity property: no strictly weaker local property suffices to ensure global atomicity [22, 23].

3.4. Online Hybrid Atomicity

Our algorithm is pessimistic: it permits an active transaction to commit whenever it is not executing an operation. The notion of *online hybrid atomicity* captures this property.

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. Now, if H is a history, define *Known*(H) = Precedes(H) $\cup$ TS(H). Known(H|X) captures what X "knows" about the timestamp order on all transactions, both committed and active. Each object must then be prepared for active transactions to choose timestamps in any order consistent with the object's local knowledge. Thus, we say that a history H is *online hybrid atomic at X* if, for every commit set C for H, and for every total order T consistent with Known(H), H|C is serializable in the order T. H is *online hybrid atomic* if, for all objects X, H is online hybrid atomic at X.

The following lemma is immediate:

Lemma 2: If H is online hybrid atomic, H is also hybrid atomic.
The algorithm proposed in this paper guarantees online hybrid atomicity.

The queue history shown earlier is hybrid atomic; in fact, each of its prefixes is online hybrid atomic. In a prefix in which either P or Q does not commit, Known(H) is empty, but the history is serializable in either order (P followed by Q or Q followed by P). Once P and Q commit, Known(H) contains the pair (Q,P). Once R executes an operation, Known(H) also contains the pairs (P,R) and (Q,R), and thus defines a total order Q-P-R on the three transactions; for a prefix containing one of R's operations to be online hybrid atomic, it needs to be serializable in the order Q-P-R, which, as argued earlier, it is.

4. Conflicts and Concurrency

This section describes our criteria for lock conflict. We begin with an informal overview of the locking protocol itself, and then we present a formal definition of our notion of dependency. We conclude with a series of examples illustrating how dependency applies to a variety of common data types.

4.1. Overview

Our protocol uses an approach similar to typical locking protocols: an operation determines whether it can proceed based on whether other active transactions have executed conflicting operations. However, our notion of "conflicts" is less restrictive than in previous work; in addition, unlike most previous work we describe precisely how commits and aborts of transactions are handled.

The protocol maintains three components for each object.

- Each transaction has an *intentions list* consisting of the sequence of operations to be applied to the object if the transaction commits. (As defined earlier, each operation consists of an invocation, argument values, termination condition, and result values.)
- The object's *committed state* reflects the effects of committed transactions. For now, it is convenient to treat the committed state as if it were simply the intentions lists for the committed transactions, arranged in timestamp order. In section 6, we describe a more compact and efficient representation.
- A set of *locks* associates each operation with the set of active transactions that have executed that operation. Locks are related by a symmetric *conflict relation* whose properties are discussed in the next section. We allow the lock conflict relation to take arguments and results into account, although it is not forced to do so.

When a transaction invokes an operation, it first constructs a *view* by appending its own intentions list to the committed state. It then chooses a result consistent with the view. Before appending the new operation to its intentions list, however, the transaction requests a lock for the operation. If another active transaction holds a conflicting lock, the lock request is refused, the result is discarded, and the invocation is later retried. (The invocation may return a different result when it is retried.) If the lock is granted, the operation is appended to the transaction's intentions list and the response is returned. (If the lock conflict relation being used does not take results into account, the lock can be requested before choosing the response.) When a transaction commits, its intentions list is merged into the committed state in timestamp order. When a transaction commits or aborts, its locks are released and its intentions list is discarded.

As an example, consider the history involving a FIFO queue shown earlier. As shown below in Section 4.3, enqueue operations on a FIFO queue need not conflict. Thus, our protocol allows concurrent enqueues, and in particular allows the history shown earlier. The order in which concurrently enqueued items should be dequeued is determined by the commit timestamps chosen by the concurrent transactions. Notice that enqueues do not commute, so commutativity-based protocols would not allow the same history.

4.2. Conflicts and Concurrency

The basic constraint governing lock conflicts is the notion of *dependency*: operations p and q cannot execute concurrently if one depends on the other. Let $\mathcal{R}$ be a binary relation between operations, and let h be an operation sequence.

Definition 3: A binary relation $\mathcal{R}$ on operations is a *dependency relation* for *Serial* if for all operation sequences h and k, and all operations p, such that

1. $h \bullet k$ and $h \bullet p$ are legal, and
2. for all q in k, $(q, p) \notin \mathcal{R}$

$h \bullet p \bullet k$ is legal.

In other words, if k is legal after h, p is legal after h, and no operation in k "depends on" p, then it should be legal to do k after p.

A dependency relation $\mathcal{R}$ is *minimal* if there is no $\mathcal{R}' \subset \mathcal{R}$ that is also a dependency relation. We will see that an object may have several distinct minimal dependency relations. We prove in Section 5 that our protocol is correct if and only if the lock conflict relation is a symmetric dependency relation.

The following lemmas describe some important properties of dependency relations.

Lemma 4: Let $\mathcal{D}$ be a dependency relation, h an operation sequence, and k_1 and k_2 operation sequences such that $h \bullet k_1$ and $h \bullet k_2$ are both legal. If no operation in k_1 depends on an operation in k_2 (i.e., for all q_1 in k_1 and q_2 in k_2, $(q_1, q_2) \notin \mathcal{D}$), then $h \bullet k_2 \bullet k_1$ is legal.

Proof: By induction on the length of k_2. The result is immediate if k_2 is empty. For the induction step, assume that $k_2 = k_2' \bullet p$, and that the theorem holds for all sequences shorter than k_2. The sequence $h \bullet k_2'$ is legal as a prefix of $h \bullet k_2$, $h \bullet k_2' \bullet p$ is legal by hypothesis, and $h \bullet k_2' \bullet k_1$ is legal by the induction hypothesis. Since no operation in k_2 depends on p, $h \bullet k_2' \bullet p \bullet k_1 = h \bullet k_2 \bullet k_1$ is legal by Definition 3. □

Definition 5: A subsequence g of h is *$\mathcal{R}$-closed* if whenever g contains an operation q of h it also contains every earlier operation p such that $q \mathcal{R} p$.

Definition 6: A subsequence g of h is a *$\mathcal{R}$-view* of h for q if g is $\mathcal{R}$-closed, and if it includes every p in h such that $q \mathcal{R} p$.

The next lemma is the key to proving the correctness of our algorithm. It says that to determine whether an operation is legal after a sequence of operations, it suffices to test whether it is legal after a subsequence that constitutes a $\mathcal{R}$-view for the operation.

Lemma 7: Let $\mathcal{R}$ be a dependency relation for *Serial*, and g and h sequences in *Serial* such that g is a $\mathcal{R}$-view of h for an operation q. If $g \bullet q$ is in *Serial*, so is $h \bullet q$.

Proof: We show by induction on the number of operations in h but not in g that $h \bullet q$ is legal. If $h = g$, the result is immediate. Assume g is missing at least one operation of h, and assume the result for views missing fewer operations. Let $h = h_1 \bullet p \bullet h_2$, where p is the first operation in h but not in g. Let $g = h_1 \bullet g_2$, and $g' = h_1 \bullet p \bullet g_2$.

The sequence $h_1 \bullet p$ is legal as a prefix of h, and $h_1 \bullet g_2 \bullet q = g \bullet q$ is legal by hypothesis. Since g is a $\mathcal{R}$-view of h for q, no operation in $g_2 \bullet q$ depends on p, thus $h_1 \bullet p \bullet g_2 \bullet q = g' \bullet q$ is legal by Definition 3.

Since g' is a $\mathcal{R}$-view of h for q, $g' \bullet q$ is legal, and g' is missing fewer operations of h than g, it follows from the induction hypothesis that $h \bullet q$ is legal. □

4.3. Examples

The definition of a dependency relation given in the previous section is not constructive: it merely gives a test for whether a given relation is a dependency relation. In this section we describe one way of deriving dependency relations more systematically from the serial specifications for objects, and give some examples of dependency relations for particular types of objects.

One way of defining a dependency relation for an object is to say that an operation depends on any earlier operations that might invalidate it. More precisely,

Definition 8: Operation p *invalidates* operation q if there exist operation sequences h_1 and h_2 such that $h_1 \bullet p \bullet h_2$ and $h_1 \bullet h_2 \bullet q$ are legal, but $h_1 \bullet p \bullet h_2 \bullet q$ is not.

Definition 9: Define the relation *invalidated-by* to contain all pairs (q,p) such that p invalidates q.

The following theorem shows that this definition yields a dependency relation:

Theorem 10: *Invalidated-by* is a dependency relation.

Proof: If not, then there exist sequences h and k and an operation p such that $h \bullet p$ and $h \bullet k$ are legal, no operation in k is invalidated by p, but $h \bullet p \bullet k$ is illegal. Let $h \bullet p \bullet k' \bullet q$ be the shortest illegal prefix of $h \bullet p \bullet k$. The sequence $h \bullet k' \bullet q$ is legal as a prefix of $h \bullet k$, $h \bullet p \bullet k'$ is legal by construction, but $h \bullet p \bullet k' \bullet q$ is illegal, hence q is invalidated by p, a contradiction. □

While *invalidated-by* is a dependency relation, it need not be a minimal dependency relation.

The remainder of this section describes dependency relations for certain simple objects, illustrating how the notion encompasses partial operations, non-deterministic operations, and operations' responses. We caution the reader not to confuse dependency relations and conflict relations. Dependency relations need not be symmetric; the conflict relations used in our algorithm, however, must be symmetric. A conflict relation will typically be constructed by taking the symmetric closure of a dependency relation.

A *File* provides *Read* and *Write* operations:

```
Read = Operation() Returns (Value)
Write = Operation(Value)
```

where Read returns the most recently written value. The unique minimal dependency relation for File objects is shown in Figure 4-1, where an entry indicates that the row operation depends on the column operation when the indicated condition holds. This relation is the *invalidated-by* relation for a File object. In this example, a read operation depends on a write operation when their argument values are distinct. Note that write operations do not depend on one another. Thus, our protocol can allow concurrent writes; when this happens, later transactions will read the value written by the transaction with the later commit timestamp. Our protocol thus encompasses and generalizes the *Thomas Write Rule* [21].

	Read(),v	Write(v),Ok
Read(),v'		$v \neq v'$
Write(v'),Ok		

Figure 4-1: Minimal Dependency Relation for File

A FIFO Queue object has two operations, *Enq* and *Deq*, where Enq places an item at the end of a queue, and Deq removes and returns the item from the front of the queue. (If the queue is empty, Deq blocks; thus, its specification is *partial*.) FIFO Queues have two distinct minimal dependency relations, shown in Figures 4-2 and 4-3. The corresponding conflict relations (obtained by taking the symmetric closures of the dependency relations) impose incomparable constraints on concurrency. In Figure 4-2, which is the *invalidated-by* relation for FIFO Queues, a Deq operation involving a given item depends on both Enq operations involving different items and Deq operations involving the same item, implying that Deq cannot execute concurrently with other Enq or Deq operations, but Enq operations can execute concurrently. In Figure 4-3, Enq operations involving different items depend on one another, and Deq operations involving the same items depend on one another, but Deq operations do not depend on Enq operations, and vice-versa. (It may seem counter-intuitive that Deq operations do not need to depend on Enq operations; however, it should become clear when we present our protocol why this is so.) With the dependency relation in Figure 4-3, an enqueuing transaction can execute concurrently with a dequeuing transaction as long as the latter can dequeue items enqueued by committed transactions.

	Enq(v),Ok	Deq(),v
Enq(v'),Ok		
Deq(),v'	$v \neq v'$	v=v'

Figure 4-2: First Minimal Dependency Relation for Queue

	Enq(v),Ok	Deq(),v
Enq(v'),Ok	$v \neq v'$	
Deq(),v'		v=v'

Figure 4-3: Second Minimal Dependency Relation for Queue

Constraints on concurrency can often be relaxed by introducing non-determinism into sequential specifications. A *Semiqueue* provides *Ins* and *Rem* operations:

```
Ins = Operation(Item)
Rem = Operation() Returns(Item)
```

Ins inserts an item in the Semiqueue, and Rem non-deterministically removes and returns an item from the Semiqueue. Like Deq, Rem returns only when there is an item to remove. (There may be an additional probabilistic guarantee, not captured by our functional specifications, that the item removed is likely to be the oldest one.) A Semiqueue object has the unique minimal dependency relation shown in Figure 4-4. This dependency relation prevents Rem operations that return the same items from executing concurrently, but allows Ins operations to execute concurrently with Rem operations, and with one another.

	Ins(v), Ok	Rem(), v
Ins(v'), Ok		
Rem(), v'		v = v'

Figure 4-4: Minimal Dependency Relation for SemiQueue

An *Account* provides *Credit*, *Post*, and *Debit* operations:

```
Credit = Operation(Dollar)
Post = Operation(Percent)
Debit = Operation(Dollar) Signals (Overdraft)
```

Credit increments the account balance by a specified amount. Post posts interest; for example, [Post(5),Ok] multiplies the account balance by 1.05. Debit attempts to decrement the balance. If the amount to be debited exceeds the balance, the operation returns with an exception, leaving the balance unchanged. Account has a unique minimal dependency relation shown in Figure 4-5. As in several of the previous examples, this relation is the *invalidated-by* relation for Account objects. An interesting aspect of this relation is that it enhances concurrency by taking operations' responses into account. For example, Credit locks need not conflict with locks for successful debits, although they must conflict with locks for attempted overdrafts, because increasing the account balance cannot invalidate a successful debit, but it can invalidate an *Overdraft* exception. If both kinds of debit operations were treated alike, debits and credits would have to be mutually exclusive, a significant cost if attempted overdrafts were infrequent. An example implementation of Account appears in the appendix.

	Credit(n),Ok	Post(n),Ok	Debit(n),Ok	Debit(n),Overdraft
Credit(m),Ok				
Post(m),Ok				
Debit(m),Ok			true	
Debit(m),Overdraft	true	true		

Figure 4-5: Minimal Dependency Relation for Account

5. A Hybrid Locking Protocol

This section presents a formal description of our locking protocol, together with its proof of correctness. The description here is designed to emphasize the general strategy followed by the protocol, and to highlight the differences with other locking protocols. In section 6, we discuss some issues that arise when designing an efficient implementation of this protocol for a particular data type. In the appendix, we present an example implementation of an Account object, illustrating how properties of the data type can be used to design efficient implementations.

Given the serial specification *Serial* of an object X, the protocol described below ensures that all histories generated by the implementation of X are hybrid atomic. For ease of exposition, we will not refer specifically to X unless necessary; thus, when we refer to an "operation" we mean an operation of X, and similarly for events.

5.1. The Protocol

A ***state machine*** is an automaton given by a set of states, a set of transitions, an initial state, and a partial transition function that 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. We say that a sequence of transitions is *accepted* by a 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.

The protocol is described by a state machine LOCK whose language consists of a set of event sequences. The machine uses a particular conflict relation, *Conflict*, to test whether one operation conflicts with another. We assume that *Conflict* is symmetric. To describe the protocol, however, we do not need to make any other assumptions about the conflict relation used by the protocol. In the next section we will show that conflict relations derived from dependency relations are both necessary and sufficient to ensure correctness of the implementation, in the sense that every complete history in L(LOCK) is hybrid atomic.

A state s of LOCK consists of four components: s.pending, s.intentions, s.committed, and s.aborted. *S.pending* is a partial function from transactions to invocation events. *S.intentions* is a total function from transactions to sequences of operations. *S.committed* is a partial function from transactions to timestamps. *S.aborted* is a set of transactions.

S.pending records pending invocations for transactions. Since each transaction is initially quiescent, s.pending is undefined for all transactions in the initial state of LOCK. S.intentions records the sequence of operations executed by each transaction. In the initial state of LOCK, s.intentions maps each transaction to the empty sequence. There are no "locks" recorded explicitly in this formal model of the algorithm; instead, the set of locks held by a transaction is implicit in the transaction's intentions list. S.committed allows us to tell which transactions have committed, and for each committed transaction records its timestamp. S.committed is initially undefined for all transactions. S.aborted records the set of transactions that have aborted, and is initially empty.

If s is a state of LOCK, define s.completed to be s.aborted $\cup$ {P | s.committed(P) $\neq \perp$}; s.completed thus consists of all transactions that have either committed or aborted. If Q $\notin$ s.completed, define View(Q,s) to be the operation sequence obtained by concatenating the intentions lists for all committed transactions in timestamp order, and then appending the intentions list for Q.[4]

The transitions of LOCK are the events involving X; their preconditions and postconditions are described below. For brevity, we assume that all input histories are well-formed. (Well-formedness could be checked explicitly by adding more state components and preconditions.) In the descriptions, the expression m[a $\rightarrow$ b], where m is a (possibly partial) function from domains A to B, a$\in$A, and b$\in$B, denotes the function identical to m except at a, which it maps to b.

In describing transitions, we write preconditions and postconditions for events, using the convention that s' denotes the state before the indicated event, and s denotes the state after the event. In addition, a state component that is not mentioned in the postcondition for an event is assumed to be unchanged by the occurrence of that event.

Invocation, commit, and abort events are inputs controlled by the transactions; thus, their preconditions are *True*. The transition for each event is quite simple: the event is simply recorded in the state of LOCK.

```
<i,X,Q>
        Postcondition:
                s.pending = s'.pending[Q → i]

<commit(t),X,Q>
        Postcondition:
                s.committed = s'.committed[Q → t]

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

Response events are somewhat more complicated:

[4] In general, View(Q,s) is not a sequence, since the set of committed transactions could be infinite. In every reachable state, however, only finitely many transactions have committed, so View(Q,s) is well-defined.

```
<r,X,Q>
    Precondition:
        $s'.pending(Q) \neq \perp$
        $Q \notin s'.completed$
        Let $q = <s'.pending(Q),r>$
        $View(Q,s') \bullet q \in$ *Serial*
        for all transactions $P \notin s'.completed \cup \{Q\}$,
            and for all operations p in $s'.intentions(P)$,
                $<p,q> \notin$ *Conflict*

    Postcondition:
        $s.pending = s'.pending[Q \rightarrow \perp]$
        $s.intentions = s'.intentions[Q \rightarrow s'.intentions(Q) \bullet q]$
```

To return a response to a transaction, there are several requirements. First, the transaction must have a pending invocation. Second, the transaction must not have already completed. Third, the operation (consisting of the <invocation, result> pair) must be legal in the transaction's "view." Finally, the operation must not conflict with any other operation already executed by another active transaction. If all these requirements are met, the response event can occur, causing the pending invocation to be removed from the state and the intentions list for the transaction to be updated to record the new operation.

Notice that s.intentions is retained for all transactions, including committed transactions. Thus, the "committed state" is simply the intentions lists for the committed transactions, arranged in timestamp order. This approach is clearly not practical. Nevertheless, it permits us to describe the protocol in a simple and general manner. All other recovery methods seem to be special cases of this use of intentions lists, in the sense that they record no more information about the past in the state. In addition, some other recovery methods seem to require restricting concurrency more than is needed for intentions lists. In later sections, we will show that there are simple optimizations that can be used in real implementations that make it possible to discard intentions lists for committed transactions.

5.2. Correctness Proof

We wish to prove the following theorem:

Theorem 11: If *Conflict* is a dependency relation, then every history in L(LOCK) is hybrid atomic.

We will show that if *Conflict* is a dependency relation, then every history in L(LOCK) is online hybrid atomic at X. Given Lemma 2, this suffices to prove Theorem 11.

We start with a simple lemma relating the state of LOCK after a history to the events in the history; the proof involves a simple induction on the length of histories in L(LOCK), and is omitted.

Lemma 12: Let H be a history in L(LOCK), let s be the state of LOCK after H, and let Q be a transaction. The following properties hold:

- $OpSeq(H|Q) = s.intentions(Q)$.
- $OpSeq(H|Q)$ ends in the invocation event $<i,X,Q> \Leftrightarrow s.pending(Q) = i$.
- $<commit(t),X,Q>$ appears in H $\Leftrightarrow s.committed(Q) = t$.
- $aborted(H) = s.aborted$.

The next lemma shows that active transactions do not conflict.

Lemma 13: Let H be a history in L(LOCK), and let s be the state of LOCK after H. If $P \neq Q$, $P \notin Completed(H)$, and $Q \notin Completed(H)$, then no operation in $s.intentions(P)$ conflicts with an operation in $s.intentions(Q)$.

Proof: An easy induction on the length of H. □

The next lemma shows a basic property of two-phase locking. It says that if two transactions are concurrent (neither commits before the other executes an operation), then there are no lock conflicts between them.

Lemma 14: Let H be a history in L(LOCK). If P and Q are transactions such that P ≠ Q, P ∉ Aborted(H), Q ∉ Aborted(H), (P,Q) ∉ Precedes(H), and (Q,P) ∉ Precedes(H), then no operation in OpSeq(H|P) conflicts with an operation in OpSeq(H|Q).

Proof: We make use of the previous lemma. Let G be the largest prefix of H that does not contain a commit event for P or Q, and let s be the state of LOCK after G. Neither P nor Q is in Completed(G). Therefore, by Lemma 13, no operation in s.intentions(P) conflicts with an operation in s.intentions(Q). By Lemma 12, OpSeq(G|P) = s.intentions(P) and OpSeq(G|Q) = s.intentions(Q), and consequently no operation in OpSeq(G|P) conflicts with an operation in OpSeq(G|Q).

We now claim that OpSeq(G|P) = OpSeq(H|P) and OpSeq(G|Q) = OpSeq(H|Q). Since no operation in OpSeq(G|P) conflicts with an operation in OpSeq(G|Q), this suffices to prove the lemma. We show the claim by contradiction. We consider P; the proof for Q is symmetric. Suppose OpSeq(G|P) ≠ OpSeq(H|P). Then OpSeq(H|P) is longer than OpSeq(G|P); let <i,r> be the first operation that occurs in OpSeq(H|P) that does not occur in OpSeq(G|P). It follows that the event <r,X,P> occurs in H and not in G. Furthermore, <r,X,P> must occur in H after a commit event for either P or Q, since G is the largest prefix of H that does not contain a commit event for either P or Q. The event <r,X,P> cannot occur after a commit event for P, since H is well-formed; therefore, it occurs after a commit event for Q. This implies, however, that (Q,P) ∈ Precedes(H), which contradicts one of the hypotheses of the lemma. □

The next lemma is needed to show that View(Q,s) contains enough information to compute the result of an operation.

Lemma 15: Let H be a history in L(LOCK), and let s_H be the state of LOCK after H. Let C be a commit set for H, and let P be an active transaction in C, i.e., P ∈ C − Committed(H). Finally, let T be a total order on transactions consistent with Known(H) such that (Q,P) ∈ T for every Q ∈ Committed(H). Then View(P,s_H) is a *Conflict*-closed subsequence of OpSeq(Serial(H|C,T)).

Proof: We first argue that View(P,s_H) is a subsequence of OpSeq(Serial(H|C,T)); we then show that it is *Conflict*-closed.

View(P,s_H) is constructed by appending s_H.intentions(Q), for each Q in Committed(H), indexed in the order given by s_H.committed(Q), and then appending s_H.intentions(P). By Lemma 12, s_H.intentions(R) = OpSeq(H|R) for every transaction R, and the order given by s_H.committed(Q) is the same as TS(H). Thus, View(P,s_H) = OpSeq(H|Q_1) • ... • OpSeq(H|Q_n) • OpSeq(H|P), where Q_1,...,Q_n are the transactions in Committed(H) in the order specified by TS(H). Since T is consistent with TS(H) and (Q,P) ∈ T for every Q ∈ Committed(H), the operations in View(P,s_H) appear in the same order in OpSeq(Serial(H|C,T)). Thus, View(P,s_H) is a subsequence of OpSeq(Serial(H|C,T)).

Now we show that View(P,s_H) is a *Conflict*-closed subsequence of OpSeq(Serial(H|C,T)). We proceed by induction on the length of H. The basis case, when H = Λ, is immediate.

For the induction step, suppose H ≠ Λ, and assume that the theorem holds for all histories in L(LOCK) that are shorter than H. Then H = K • *e* for some history K in L(LOCK) and some event *e*. Let s_K be the state of LOCK after K. By induction, the lemma holds for K and s_K.

First, note that Precedes(K) ⊆ Precedes(H), TS(K) ⊆ TS(H), Committed(K) ⊆ Committed(H), and Aborted(K) ⊆ Aborted(H). Thus, C and T satisfy the conditions of the lemma for K. By induction, View(P,s_K) is a *Conflict*-closed subsequence of OpSeq(Serial(K|C,T)). There are three cases to consider, depending on the type of *e*.

1. Suppose *e* is an invocation or abort event for some transaction R. Then s_H.intentions = s_K.intentions, and s_H.committed = s_K.committed. Thus, View(P,s_H) = View(P,s_K). If *e* is an abort event, R ∉ C. Otherwise, note that OpSeq throws away pending invocation events. In either

case, OpSeq(Serial(H|C,T)) = OpSeq(Serial(K|C,T)). The result follows from the induction hypothesis.

2. Suppose *e* is a commit event for some transaction R. Since OpSeq throws away commit and abort events, OpSeq(Serial(H|C,T)) = OpSeq(Serial(K|C,T)). In addition, View(P,s_H) is obtained from View(P,s_K) by inserting OpSeq(H|R) in the position determined by the timestamp for R. To show that View(P,s_H) is a *Conflict*-closed subsequence of OpSeq(Serial(H|C,T)), we must show that for every operation in View(P,s_H), every earlier conflicting operation from OpSeq(Serial(H|C,T)) is also in View(R,s_H). By the induction hypothesis, this is true for operations that are in both View(P,s_H) and View(P,s_K). The operations that are in View(P,s_H) but not in View(P,s_K) are the operations in s_H.intentions(R). Suppose an operation *r* in OpSeq(Serial(H|C,T)) precedes some operation *q* in s_H.intentions(R) and conflicts with it, and let S be the transaction that executed *r*. Then by Lemma 14 and the constraints on T, <S,R> ∈ Precedes(H). By the definition of Precedes(H), S ∈ Committed(H), and thus *r* appears in View(P,s_H).

3. Finally, suppose that *e* is a response event <r,X,R>. If P = R the result follows from the induction hypothesis and the precondition for *e*, since the operation added to s_H.*intentions*(P) by *e* cannot conflict with operations executed by active transactions, and all other operations in OpSeq(Serial(H|C,T)) are in View(P,s_H).

   If P and R are distinct, then View(P,s_H) = View(P,s_K). If R ∉ C, then OpSeq(Serial(H|C,T)) = OpSeq(Serial(K|C,T)), and the result follows by induction. Otherwise, OpSeq(Serial(H|C,T)) differs from OpSeq(Serial(K|C,T)) in that it contains an extra operation for R. Since H is well-formed, R ∉ Committed(H). By Lemma 14, an operation executed by R conflicts with an operation executed by another transaction S only if <S,R> ∈ Precedes(H). Therefore, every operation in OpSeq(Serial(H|C,T)) that conflicts with an operation *q* executed by R appears before *q*. The result follows by induction.

□

Now we are ready to prove the main result.

Theorem 16: Suppose H is a history in L(LOCK), and suppose *Conflict* is a dependency relation. Then H is online hybrid atomic at X.

Proof: The proof proceeds by induction on the length of H. The basis case, when H = Λ, is immediate.

For the induction step, suppose H ≠ Λ, and assume the result for all histories in L(LOCK) shorter than H. Then H = K • *e* for some history K in L(LOCK) and some event *e*. Since K is shorter than H, the theorem holds for K.

Note that all events in H (and K) involve only X; thus, H = H|X.

Let s_K be the state of LOCK after K.

Now, let C be a commit set for H, and let T be a total order on transactions consistent with Known(H). To show that H is online hybrid atomic at X, it suffices to show that H|C is serializable in the order T, *i.e.*, that OpSeq(Serial(H|C,T)) is legal.

First, note that Precedes(K) ⊆ Precedes(H), TS(K) ⊆ TS(H), Committed(K) ⊆ Committed(H), and Aborted(K) ⊆ Aborted(H). Thus, C and T satisfy the conditions of the definition of on-line hybrid atomicity for K. There are now two cases, depending on the type of *e*.

1. Suppose *e* is a commit, abort, or invocation event. Note that OpSeq throws away pending invocation events, commit events, and abort events. Thus, OpSeq(Serial(H|C,T)) = OpSeq(Serial(K|C,T)). Since the theorem holds for K, it also holds for H.

2. Suppose that *e* is a response event <r,X,P>. This is the difficult case. Note that if P ∉ C then H|C = K|C, and the result follows from the induction hypothesis. So assume that P ∈ C.

   Let *Active* = C − Committed(H). The sequence OpSeq(Serial(H|C,T)) can be written as h_1 • OpSeq(H|P) • h_2, where h_1 = OpSeq(serial(H|C_1)) and h_2 = OpSeq(serial(H|C_2)), and C_1 and C_2

are chosen such that Committed(H) $\subseteq C_1$, $C_2 \subseteq$ Active, and P $\notin C_1 \cup C_2$. The sequences h_1 and h_2 respectively contain the operations of transactions ordered before and after P by T. C_1 contains Committed(H) because T is consistent with Precedes(H), and since e = <r,X,P>, it follows from the definition of Precedes(H) that (Q,P) $\in$ Precedes(H) for all Q $\in$ Committed(H). Note that h_i = OpSeq(Serial(K|C_i,T)), since P $\notin C_i$.

Since P is executing a response event, and H is well-formed, no commit event for P appears in H. Also, C_2 is chosen such that $C_2 \subseteq$ Active; thus, if Q $\in C_2$, no commit event can appear in H for Q. It is an immediate consequence of the definitions that P and Q are unrelated by Precedes(H). Thus, by Lemma 14, no operation in OpSeq(H|P) conflicts with an operation in OpSeq(H|Q) for any Q $\in C_2$. By the definition of h_2, no operation in OpSeq(H|P) conflicts with an operation in h_2.

We show that $h_1 \bullet h_2$ and $h_1 \bullet$ OpSeq(H|P) are both legal. Since no operation in OpSeq(H|P) conflicts with an operation in h_2, it then follows from Lemma 4 that $h_1 \bullet$ OpSeq(H|P) $\bullet h_2$ is also legal, giving the desired result.

To show that $h_1 \bullet h_2$ is legal, we note that it is simply OpSeq(Serial(H|C – {P},T)), which is the same as OpSeq(Serial(K|C – {P},T)). By the induction hypothesis, this sequence is legal.

To show that $h_1 \bullet$ OpSeq(H|P) is legal, note that OpSeq(H|P) = OpSeq(K|P) $\bullet$ <i,r> for some invocation i. By induction, $h_1 \bullet$ OpSeq(K|P) is legal, since it equals OpSeq(serial(K|$C_1 \cup$ {P},T)). By the precondition for e, <i,r> does not conflict with any operation in OpSeq(H|Q) for any Q $\in$ Active; thus, View(P,s_K) contains all operations of $h_1 \bullet$ OpSeq(K|P) with which <i,r> conflicts.

Since e is a response event for P, (Q,P) $\in$ Precedes(H) for all Q $\in$ Committed(H). Since T is consistent with Known(H), K, C, and T satisfy the hypotheses of Lemma 15, which then implies that View(P,s_K) is a *Conflict*-closed subsequence of $h_1 \bullet$ OpSeq(K|P). By the precondition for e, View(P,s_K) and View(P,s_K) $\bullet$ <i,r> are both legal, hence by Lemma 7, so is $h_1 \bullet$ OpSeq(K|P) $\bullet$ <i,r> = $h_1 \bullet$ OpSeq(H|P).

□

The theorem above shows that a sufficient condition for LOCK to be correct is that the conflict relation be a dependency relation. We now show that this is also a necessary condition.

Theorem 17: If the conflict relation used in LOCK is not a dependency relation for *Serial*, then L(LOCK) contains a history that is not online hybrid atomic.

Proof: If the conflict relation is not a dependency relation, choose sequences h and k and an operation p such that $h \bullet p$ and $h \bullet k$ are legal, no operation in k conflicts with p, and $h \bullet p \bullet k$ is not legal. Consider the following scenario. Transaction P executes the operations in h and commits, Q executes p, and R executes the operations in k. By hypothesis, p does not conflict with any operations executed by R. If Q commits with a lower timestamp than R, the accepted history is not serializable in timestamp order. □

6. Compaction

Although the use of intentions lists facilitates our proofs, it has the disadvantage that object representations are neither compact nor efficient. For example, the size of a Queue representation has no relation to the number of items present in the queue, and the item at the head of the queue must be found by a linear search. These problems can be alleviated by replacing intentions lists with more compact and efficient representations. For example, we can replace a sequence of operations with the state (or *version*) that results from applying those operations to the initial state. For a Queue or Semiqueue, a version might be represented by an array or a linked list, while for an Account an integer cell might be used.

In this section, we describe a general technique for discarding intentions lists for committed transactions, replacing them with the version that represents their net effect. Each object keeps track of an operation sequence that forms a prefix for every view that will henceforth be assembled by any transaction. Each view is assembled by appending some sequence of intentions lists to the common prefix. When a committed transaction is sufficiently old, it can be "forgotten" by appending its intentions list to the common prefix, discarding both its intentions list and its commit timestamp. This common prefix is represented compactly as a version.

It is important to realize that a transaction cannot necessarily be forgotten as soon as it commits, because intentions lists must be appended to the common prefix in commit timestamp order, but commit events for concurrent transactions need not occur in timestamp order. Instead, care must be taken to ensure that a transaction is forgotten only when no active transaction can commit with an earlier timestamp. To recognize when it is safe to forget a transaction, we introduce some auxiliary components to our state machine. *S.clock* keeps track of the latest observed commit timestamp; it has an initial value of $-\infty$. *S.bound* is a partial function from transactions to commit timestamps, initially undefined for all transactions. If Q is an active transaction, s.bound(Q) is a lower bound on the possible commit timestamps that Q could choose when it commits.

We add the following postconditions to the transitions for LOCK:

```
<i,X,Q>
        Postcondition:
                s.bound = s'.bound[Q → s.clock]

<r,X,Q>
        Postcondition:
                s.bound = s'.bound[Q → s.clock]

<commit(t),X,Q>
        Postcondition:
                s.clock = max(s'.clock,t)
                s.bound = s'.bound[Q → ⊥]
<abort,X,Q>
        Postcondition:
                s.bound = s'.bound[Q → ⊥]
```

These additional components have no effect on L(LOCK); they serve only for bookkeeping. The idea is that we maintain a local clock that equals the maximum of the commit timestamps for transactions that have committed at the object. Since the commit timestamp order is required to be consistent with the precedes order at each object, the lower bound on the commit timestamp for an active transaction is increased to the current clock time whenever the transaction invokes an operation or has an operation return. If Q is active and <i,X,Q> occurs, then by the constraints on commit timestamps the timestamp eventually chosen by Q must be greater than the commit timestamp for any transaction committed at X at the time that <i,X,Q> occurs, and similarly for <r,X,Q>. Thus, the current clock time when <i,X,Q> or <r,X,Q> occurs does constitute a lower bound on the commit timestamp eventually chosen by Q.

Before describing details of how intentions lists are compacted, we present some properties of s.bound and s.clock. We start with a simple lemma relating these auxiliary components of LOCK to the other components. The proof involves a simple induction on the length of histories in L(LOCK), and is omitted.

Lemma 18: Let Q be a transaction, H a history in L(LOCK), and s the state of LOCK after accepting H.

1. If s.bound(Q) is defined, there exists a transaction P such that s.committed(P) = s.bound(Q).
2. If s.committed(Q) is defined, s.committed(Q) $\leq$ s.clock.
3. If s.committed(Q) is undefined and s.intentions(Q) $\neq \Lambda$, then s.bound(Q) is defined.

The following lemma describes how s.bound and s.committed give information about Known(H); in particular, it (together with the first part of Lemma 18) shows that s.bound(Q) is a lower bound on Q's eventual commit timestamp.

Lemma 19: Let P and R be transactions, H a history in L(LOCK), and s the state of LOCK after accepting H. If s.bound(R) and s.commited(P) are defined and s.commited(P) $\leq$ s.bound(R), then (P,R) $\in$ Known(H).

Proof: By induction on the length of H. The result is immediate when H is empty. For the induction step, let H = G • *e*, where *e* is a single event, and let s' be the state after G, and s the state after H. Fix a pair of transactions P and R. If *e* is associated with any transaction other than P or R, the values of s.bound(R) and s.committed(P) are unaffected. The result holds vacuously if *e* is an abort, invocation, or response for P, because s.committed(P) is undefined. The result is also vacuous if *e* is an abort or commit for R, because s.bound(R) is undefined. If *e* is an invocation or response for R, then s.bound(R) = s.clock $\geq$ s.committed(P), by Lemma 18. Moreover, (P,R) $\in$ Known(H), since R executed an invocation or response after P committed. Suppose *e* is <commit(t),X,P>. If *t* > s.bound(R), the result holds vacuously. Otherwise, by Lemma 18, there exists a transaction Q such that s.committed(Q) = s.bound(R). By the induction hypothesis, (Q,R) $\in$ Known(G), and since Known(G) $\subseteq$ Known(H), (Q,R) $\in$ Known(H). Suppose s.committed(P) = s.committed(Q); then P = Q (by well-formedness), and by induction (P,R) $\in$ Known(H). Otherwise, s.committed(P) < s.committed(Q), so (P,Q) $\in$ TS(H), and thus by transitivity we have that (P,R) $\in$ Known(H). □

Now we describe how intentions lists are compacted. Let s be a state of LOCK. Informally, the horizon time for s is a lower bound on the commit timestamp that can be chosen by an active transaction. The result of concatenating the intentions lists of all transactions whose commit timestamps precede the horizon time is certain to be a prefix of every transaction's view, and thus can be compacted into a version. More formally:

Definition 20: s.horizon = max($-\infty$, min(min{s.bound(P) | s.bound(P) $\neq \perp$},
max{s.committed(P) | s.committed(P) $\neq \perp$}))

In other words, the horizon time is either $-\infty$ (if there are no active or committed transactions), or it is the smaller of the smallest bound for an active transaction and the largest commit timestamp for a committed transaction. If there are no active transactions, then the horizon timestamp is the largest commit timestamp. If there is an active transaction, however, all we know about its eventual commit timestamp is that it will be bigger than the recorded lower bound for the transaction, so we should not compact intentions lists for committed transactions whose timestamps are bigger than that lower bound.

Let $Q_1,...,Q_n$ be the sequence of transactions for which s.commit is defined, indexed in timestamp order, and let $Q_1,...,Q_k$ be the subsequence of transactions such that s.commit(Q_i) $\leq$ s.horizon. We define the following "auxiliary" components.

Definition 21: s.permanent = s.intentions(Q_1) • ... • intentions(Q_n)

Definition 22: s.common = s.intentions(Q_1) • ... • intentions(Q_k)

Clearly, s.common is a prefix of s.permanent. To compute the response to an invocation for Q, we need to compute View(s,Q). If Q is an active transaction, then View(s,Q) = s.permanent • s.intentions(Q), of which s.common is a prefix. Thus, s.common can be compacted into a single version. To show that this is true, we show that s.common grows monotonically.

Lemma 23: Let H = G • *e* be a history in L(LOCK), and let s_G and s_H be states of LOCK after G and H. Then s_G.common is a prefix of s_H.common.

Proof: If *e* is an invocation, response, or abort event for Q, then s_H.committed = s_G.committed, and s_H.bound(Q) either equals s_G.bound(Q), becomes larger than s_G.bound(Q), or becomes $\perp$. Regardless, s_H.horizon $\geq$ s_G.horizon. Since s_H.committed = s_G.committed, s_G.common is a prefix of s_H.common.

If *e* is a commit event for Q, there are two cases. If *e* is the first commit event for Q, so s_G.common is a prefix of s_H.common. Otherwise, s_G.common = s_H.common. □

The following theorem follows easily from the above lemma.

Theorem 24: Let G and H be histories in L(LOCK) such that G is a prefix of H, and let s_G and s_H be states of LOCK after G and H. Then s_G.common is a prefix of s_H.common.

Since s.common grows monotonically, we can represent it by keeping a version s.version and periodically computing a new version by applying (in commit timestamp order) the intentions lists for transactions P with s.bound(P) $\leq$ s.horizon to s.version.

It is not always necessary to keep explicit track of transactions' lower bounds. For example, if one operation conflicts with every other operation, as Deq does in Figure 4-2 (ignoring the argument and result values), then all committed transactions can be forgotten whenever a dequeuing transaction commits or aborts. Transaction Q may acquire a Deq lock only if no other active transaction has executed any operations, implying that s.bound(P) = $\perp$ for all P distinct from Q, and hence s.horizon = s.bound(Q). The committed state of a queue can be represented as a single committed version together with a set of intentions consisting entirely of Enq operations. Thus, the size of the representation would be proportional to the number of elements in the queue.

The Queue conflict relation shown in Figure 4-3 can also be specially optimized. Here, all operations that do not conflict commute, thus a transaction can be forgotton as soon as it commits at an object. The resulting scheme is equivalent to a commutativity-based locking scheme of Weihl [22, 25].

7. Discussion

7.1. Comparison With Commutativity-Based Schemes

As mentioned above, the precedes order captures potential "information flow" between transactions. Most mechanisms based on two-phase locking ensure that transactions are serializable in every total order consistent with precedes, a property known as *dynamic atomicity* [22, 23]. Conflict-based concurrency control mechanisms for dynamic atomicity include those proposed by Eswaran et al. [5], Korth [13], Bernstein et al. [2], and Weihl [22, 25]. These mechanisms are all based on a notion of *commutativity*. Informally, two operations commute if executing them in either order always yields the same results and the same final object state. If two operations do not commute, their locks must conflict.

We now show that "failure to commute" is a dependency relation, although not necessarily a minimal dependency relation. It follows that our protocol is less restrictive than the commutativity-based protocol cited above; our protocol can achieve at least as much concurrency. Our examples show that lock conflict relations induced by dependency may be weaker than or incomparable to those induced by the commutativity-based protocols.

We use the following notion of commutativity taken from [22], a notion that encompasses both partial and non-deterministic operations.

Definition 25: Two operation sequences h and h' are *equivalent* if they cannot be distinguished by any future computation: $h \bullet g$ is legal if and only if $h' \bullet g$ is legal for all operation sequences g.

Definition 26: Two operations p and q *commute* if for all operation sequences h, whenever $h \bullet p$ and $h \bullet q$ are both legal, then $h \bullet p \bullet q$ and $h \bullet q \bullet p$ are legal and equivalent.

Lemma 27: If $h \bullet p$ and $h \bullet k$ are legal operation sequences and p commutes with every operation in k, then $h \bullet p \bullet k$ and $h \bullet k \bullet p$ are legal and equivalent.

Proof: By induction on the number of operations in k. The result is trivial when k is empty. For the induction step, suppose $k = k' \bullet q$, where q is a single operation, and assume the result holds for k'. Then by induction, $h \bullet p \bullet k'$ is legal and equivalent to $h \bullet k' \bullet p$. By hypothesis, $h \bullet k' \bullet q$ (= $h \bullet k$) is legal. Since p and q commute, $h \bullet k' \bullet p \bullet q$ is legal and equivalent to $h \bullet k' \bullet q \bullet p$. The latter is just $h \bullet k \bullet p$. The former is equivalent to $h \bullet p \bullet k' \bullet q$, since $h \bullet k' \bullet p$ is equivalent to $h \bullet p \bullet k'$. But this is just $h \bullet p \bullet k$. □

Theorem 28: "Failure to commute" is a dependency relation.

Proof: Let *NC* denote failure to commute. Let *h* and *k* be operation sequences and let *p* be an operation such that *h*•*k* and *h*•*p* are legal, and such that for all *q* in *k*, $(q,p) \notin NC$. It suffices to show that *h*•*p*•*k* is legal. This is immediate from Lemma 27. □

Lock conflict relations induced by dependency may be weaker than or incomparable to those induced by commutativity. For example, consider an Account object. Commutativity-based protocols impose a lock conflict relation that includes (at least) the conflicts shown in Figure 7-1. This conflict relation permits strictly less concurrency than the dependency relation shown in Figure 4-5. The additional restrictions arise because the commutativity-based protocols require Post operations to conflict with Credit and Debit operations, while the dependency-based protocols do not. In the Queue example, by contrast, the commutativity-based protocols induce lock conflicts identical to those induced by the minimal dependency relation shown in Figure 4-3. Here, however, the commutativity-based protocols do not permit the incomparable conflict relation induced by the minimal dependency relation in Figure 4-2.

	Credit(n),Ok	Post(n),Ok	Debit(n),Ok	Debit(n),Overdraft
Credit(m),Ok		true		true
Post(m),Ok	true		true	true
Debit(m),Ok		true	true	
Debit(m),Overdraft	true	true		

Figure 7-1: "Failure to Commute" Relation for Account

In addition to requiring fewer conflicts than commutativity-based protocols, our work also generalizes most other work on type-specific two-phase locking by allowing the results returned by an operation to be used in choosing the appropriate lock, and by permitting operations to be partial and non-deterministic. Some other protocols (e.g., see [18]) 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 protocol shows how the results of an operation, as well as names and arguments, can be used systematically to determine the lock needed. (The commutativity-based protocols in [22, 25] also permit result information to be used in choosing locks.)

In addition, other protocols (except for those in [22, 25]) require operations to be total and deterministic. Partial operations are important for modeling producer-consumer relationships, in which one transaction is consuming data produced by another. Such situations, while perhaps uncommon in traditional database applications, are more common in general distributed or object-oriented systems. Similarly, non-deterministic operations are an important source of concurrency; compare, for example, the dependency relations for Queue and SemiQueue shown earlier. (Non-determinism can also increase availability; see [8] for an example.)

Another way in which our work differs from most other work on type-specific concurrency control is in the treatment of recovery. With the exception of [22, 25], the other work ignores recovery.

A more general form of hybrid atomicity is defined in [22, 23], permitting read-only transactions to be treated specially, as in the multi-version protocols in [3, 4, 24]. Timestamps for read-only transactions are chosen when they start, while timestamps for other transactions are chosen when they commit. This algorithm is the origin of the term "hybrid atomicity," since the protocols combine aspects of dynamic atomic protocols (such as common two-phase protocols) and static atomic protocols (such as Reed's multiversion protocol). In fact, hybrid atomicity is upward compatible with dynamic atomic protocols: dynamic atomic protocols guarantee serializability of committed transactions in all total orders consistent with Precedes(H); since TS(H) is one such order, global atomicity is still obtained when dynamic and hybrid atomic objects are combined in a single system.

Our results suggest that dependency is a more fundamental property than commutativity for understanding concurrency control for typed objects. In addition, the notion of a dependency relation arises in a variety of other related contexts.[5] The constraints on the availability realizable by quorum consensus replication [8] can be expressed in terms of dependency relations. Dependency relations also form the basis for validation in type-specific optimistic concurrency control mechanisms [9], as well as type-specific locking schemes based on multi-version timestamping [10], and schemes that provide high levels of availability in the presence of partitions [11].

To summarize, we have defined a new locking protocol that permits more concurrency than existing commutativity-based protocols. It permits operations to be both partial and non-deterministic, and it permits results of operations to be used in choosing locks. The protocol exploits type-specific properties of objects; we have shown how to define a necessary and sufficient set of constraints on lock conflicts directly from the data type specification. The protocol is optimal in the sense that no hybrid atomic locking scheme can permit more concurrency.

I. An Example Implementation

To illustrate how our locking protocol might be used in practice, this appendix describes an implementation of the Account data type using Avalon/C++ [12], a programming language that supports hybrid atomicity. We assume some familiarity with C++ [20]. Although Avalon/C++ supports nested transactions, this example assumes only a single-level transaction model.

We start by describing the subsidiary data types used by the Account implementation. Avalon programmers do not manipulate transaction timestamps directly. Instead, Avalon provides a `trans_id` data type to permit the programmer to test serialization orders at run-time.[6]

```
class trans_id: public recoverable {
 private:
  ...                                 // representation
 public:
  trans_id();                         // constructor
  bool operator==(trans_id& who);  // equal?
  bool operator<(trans_id& who);   // serialized before?
  ...                                 // other operations
};
```

A transaction generates an identifier by a call to *new*:

```
trans_id* who = new trans_id;
```

Transaction identifiers are partially ordered by the overloaded operators ">" and "<." If transactions P and Q respectively create identifiers t1 and t2, then the expression

```
t1 < t2
```

evaluates to *true* if and only if $(P,Q) \in$ Known(H), where H is the current history.

An account object maintains lock information in a *lock table*.

```
enum lock_type {CREDIT_LOCK, POST_LOCK, DEBIT_LOCK, OVERDRAFT_LOCK};
```

[5]The definition of a dependency relation given in this paper is stated differently from that in other papers by Herlihy, but is easily shown to be equivalent.

[6]Avalon/C++ defines `bool` to be an enumeration type with **TRUE** set to 1 and **FALSE** set to 0.

```
class lock_tab {
 private:
  ...                             // representation
 public:
  lock_tab();                     // constructor
  void define(lock_type mode0,    // register a lock conflict
              lock_type mode1);
  bool conflict(lock_type mode,   // ok to grant lock?
                trans_id* who);
  void grant(lock_type mode,      // give lock to caller
             trans_id* who);
  void release(trans_id* who);    // release caller's locks.
};
```

The account operations are represented by the enumeration type `lock_type`. An empty lock table is created by declaring a variable of type `lock_tab`, and the `define` operation marks two operations as conflicting. The `conflict` operation takes a lock type and a transaction identifier, and returns *true* if no other transaction holds a conflicting lock. The `grant` operation grants a lock for a specified operation, and `release` discards all locks held by a transaction.

The net effect of a transaction that executes multiple Credits, Debits, and Posts is to replace the balance *b* by the affine transformation *m*b+a* for some *m* and *a*. Each transaction's intention is recorded in the following struct:

```
struct intent {float mul; float add;
    intent(float m, float a) {mul = m; add = a;};
};
```

The last component defines a constructor operation for initializing the struct. The intention associated with each transaction is kept in a table:

```
class intent_tab {                   // map trans -> intention.
 private:
  ...                                // representation
 public:
  intent_tab();                      // constructor
  intent lookup (trans_id* who);     // return intention
  void insert (trans_id* who,        // bind trans to intention
               intent what);
  void discard (trans_id* who);      // discard intention
};
```

`Lookup` returns a transaction's current intention. If none exists, it returns an intention with multiplicative and additive components *1.0* and *0.0* respectively.

Intentions for committed transactions are discarded using the horizon time scheme described in Section 6. Each active transaction keeps track of the latest committed transaction guaranteed to be serialized before itself. This information is kept in a table:

```
class bound_tab {                    // map trans -> lower bound
 private:
  ...                                // representation
 public:
  bound_tab();                       // constructor
  void insert(trans_id* who,         // register new lower bound
              trans_id* bnd);
  void discard(trans_id* who);       // discard lower bound
  trans_id* min();                   // horizon transaction
};
```

Transactions that are committed but not yet forgotten are kept in a heap.

```
class id_heap {                    // sorted heap of transactions
```

```
  private:
    ...                             // representation
  public:
    id_heap();                      // constructor
    trans_id* top();                // return oldest transaction
    trans_id* remove();             // remove oldest transaction
    void insert(trans_id* who);     // insert transaction
    bool empty();                   // is heap empty?
};
```

This data type provides operations for creating an empty heap, inserting a transaction identifier in the heap, and observing or removing the oldest (i.e., minimal with respect to "<") identifier in the heap.

We are now ready to examine the Account implementation itself.

```
class account: public subatomic {
  private:
    lock_tab locks;                 // locks for operations
    intent_tab intentions;          // intentions list
    float bal;                      // committed balance
    id_heap committed;              // committed but unforgotten transactions
    trans_id* clock;                // most recent transaction to commit
    bound_tab bounds;               // earliest possible commit times
    void forget();                  // for forgetting committed transactions
    status sufficient(trans_id* who, // balance covers debit?
                      float amt);
  public:
    account();
    void credit(float amt);
    bool debit(float amt);
    void post(float amt);
    void commit(trans_id* who);
    void abort(trans_id* who);
};
```

The "public subatomic" declaration means that this data type ("class" in C++ terminology) inherits certain operations necessary for short-term synchronization and for ensuring that the object is recorded properly on stable storage. The object's internal representation is given by the fields following the keyword *private*. The `locks` component keeps track of the locks, `intent_tab` records transactions' intentions, `bal` is the account balance left by "forgotten" committed transactions, and `committed` keeps track of transactions that have committed but have not yet been forgotten. The internal function `forget` uses the `clock` and `bounds` fields to implement the compaction scheme described in Section 6. The internal function `sufficient` determines whether the balance covers an attempted debit by a particular transaction.

When an account is created, the *account* constructor is invoked:

```
account::account() {
  pinning() {                       // making update
    clock = new trans_id;           // clock is creator's id
    bal = 0.0;                      // zero initial balance
    // Set up lock conflicts.
    locks.define(CREDIT_LOCK, OVERDRAFT_LOCK);
    locks.define(POST_LOCK, OVERDRAFT_LOCK);
    locks.define(DEBIT_LOCK, DEBIT_LOCK);
    }
  }
```

To ensure proper crash recovery, all modifications to the object must occur inside a `pinning` statement. Most of the object's members are implicitly initialized. The clock is initialized with the creator's identifier, the balance is initialized to zero, and the lock table is initialized with the conflict relation shown in Figure 4-5.

The Credit function is implemented as follows.

```
void account::credit(float amt) {
  trans_id* who = new trans_id;                // Get caller's id.
  when (!locks.conflict(CREDIT_LOCK, who))     // Check for conflict.
    pinning() {                                // Making update.
      locks.grant(CREDIT_LOCK, who);           // Acquire lock ...
      intent i = intentions.lookup(who);       // and current intention.
      i.add = i.add + amt;                     // Record credit ...
      intentions.insert(who, i);               // and register new intention.
      bounds.insert(who, clock);               // Note new bound.
      }
  }
```

Each atomic object has an associated mutual exclusion lock, similar to a monitor lock. The when statement is similar to a guarded command. It repeatedly acquires the lock and evaluates the condition. If the condition is *true*, the associated statement is executed and the lock is released. Otherwise, the lock is released and the condition is retried after an arbitrary duration. The Credit operation generates an identifier for the caller, and checks for lock conflicts. If none is found, the caller's intention is updated, and the current clock value is recorded as the transaction's new bound. The Post operation is similar, and is omitted.

Debit is slightly more complex.

```
bool account::debit(float amt) {
  trans_id* who = new trans_id;               // Get caller's id
  whenswitch (sufficient(who, amt)){
    case YES:                                 // debit ok
      pinning() {                             // Making update.
        locks.grant(DEBIT_LOCK, who);         // Acquire lock ...
        intent i = intentions.lookup(who);   // find intention ...
        i.add = i.add - amt;                  // record debit ...
        intentions.insert(who, i);            // and register new intention.
        bounds.insert(who, clock);            // Note new bound ...
        return TRUE;                          // and return success code.
        }
    case NO:                                  // Ok to refuse debit.
      pinning() {                             // Making update.
        locks.grant(OVERDRAFT_LOCK, who);    // Acquire lock ...
        return FALSE;                         // and return overdraft code.
        }
      }
    }
```

The *whenswitch* statement is a generalization of the *when* statement that replaces the boolean expression with an expression of an enumeration type. Here, Debit calls upon the internal procedure *sufficient*, which returns *YES* if the account balance covers the debit, *NO* if the debit should be refused, and *MAYBE* if lock conflicts leave the account status ambiguous.

The code for *sufficient* appears below.

```
enum status {YES, NO, MAYBE};

status account::sufficient(trans_id* who, float amt){
  float view = bal;                 // Construct view
  id_heap h; h = committed;         // Copy heap of committed id's.
  while (!h.empty()) {              // Apply each committed intention.
    intent i = intentions.lookup(h.remove());
    view = i.mul * view + i.add;
    }
  intent i = intentions.lookup(who); // Apply caller's intention.
  view = i.mul * view + i.add;

  // Sufficient funds?
```

```
    if (view >= amt && !locks.conflict(DEBIT_LOCK, who)) return YES;

    // Insufficient funds?
    if (view < amt && !locks.conflict(OVERDRAFT_LOCK, who)) return NO;

    // Can't tell.
    return MAYBE;
  }
```

Atomic objects in Avalon/C++ provide *commit* and *abort* operations, which are called by the system when transactions commit or abort. The commit operation for Account is the following:

```
void account::commit(trans_id* who){
  when (TRUE)                         // Always ok to commit.
    pinning() {                       // Updating object.
      if (*clock < *who) clock = who; // Advance clock.
      locks.release(who);             // Release locks.
      bounds.discard(who);            // Discard bound.
      committed.insert(who);          // Mark as committed.
      forget();                       // Try to forget.
    }
}
```

The clock is advanced, the committing transaction's locks are released, its lower bound is discarded, the transaction is marked as committed. The internal function *forget* is called to forget committed transactions:

```
void account::forget(){
  trans_id* horizon = bounds.min();
  while (!committed.empty() && *(committed.top()) < *horizon) {
    trans_id* t = committed.remove(); // Remove the transaction,
    intent i = intentions.lookup(t);  // find its intention,
    bal = i.mul * bal + i.add;        // apply it,
    intentions.discard(t);            // and discard it.
  }
}
```

This function recomputes the horizon time, and applies and discards the intentions for all committed transactions serialized before the horizon.

Abort is similar to *commit*:

```
void account::abort(trans_id* who){
  when (TRUE)                         // Always ok to abort.
    pinning() {                       // Updating object.
      locks.release(who);             // Release locks.
      bounds.discard(who);            // Discard bound.
      intentions.discard(who);        // Discard intentions.
      forget();                       // Try to forget.
      }
  }
```

References

[1] P.A. Bernstein and N. Goodman.
Concurrency control in distributed database systems.
ACM Computing Surveys 13(2):185-222, June, 1981.

[2] P.A. Bernstein, N. Goodman, and M.Y. Lai.
Two-part proof schema for database concurrency control.
In *Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer networks*. February, 1981.

[3] A. Chan, S. Fox, W.T. Lin, A. Nori, and D. Ries.
The implementation of an integrated concurrency control and recovery scheme.
In *Proceedings of the 1982 SIGMOD Conference*. ACM SIGMOD, 1982.

[4] D.J. Dubourdieu.
Implementation of distributed transactions.
In *Proceedings 1982 Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 81-94. 1982.

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

[6] Goree, J. A.
Internal consistency of a distributed transaction system with orphan detection.
Master's thesis, MIT, January, 1983.
Available as MIT/LCS/TR-286.

[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] M.P. Herlihy.
A quorum-consensus replication method for abstract data types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[9] M.P. Herlihy.
Optimistic concurrency control for abstract data types.
In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 206-217. August, 1986.

[10] M.P. Herlihy.
Extending Multiversion Timestamping Protocols to Exploit Type Information.
IEEE Transactions on Computers C-35(4), April, 1987.
Special issue on parallel and distributed computing.

[11] M.P. Herlihy.
Dynamic quorum adjustment for partitioned data.
ACM Transactions on Database Systems 12(2), June, 1987.
Also available as TR CMU-CS-86-147.

[12] M.P. Herlihy and J.M. Wing.
Avalon: Language Support for Reliable Distributed Systems.
In *17th Symposium on Fault-Tolerant Computer Systems*. July, 1987.
Also available as TR CMU-CS-86-167.

[13] H.F. Korth.
Locking primitives in a database system.
Journal of the ACM 30(1), January, 1983.

[14] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[15] 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.

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

[17] D.P. Reed.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

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

[19] 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.

[20] B. Stroustrup.
The C++ Programming Language.
Addison Wesley, Reading, Mass., 1986.

[21] R.H. Thomas.
A majority consensus approach to concurrency control for multiple copy databases.
ACM Transactions on Database Systems 4(2):180-209, June, 1979.

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

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

[24] W.E. Weihl.
Distributed version management for read-only actions.
IEEE Transactions on Software Engineering SE-13(1):55-64, January, 1987.

[25] W.E. Weihl.
Commutativity-based Concurrency Control for Abstract Data Types.
In *Proceedings of the Twenty-first Annual Hawaii Conference on System Sciences.* January, 1988.
Revised version to appear in a special issue on parallel and distributed computing of IEEE Transactions on Computers.

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