(4)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-382

THE IMPACT OF RECOVERY ON CONCURRENCY CONTROL

(Extended Abstract)

William E. Weihl

February 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

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-382	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)
The Impact of Recovery on Concurrency Control

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

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	February 1989	17

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

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

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

It is widely recognized by practitioners that concurrency control and recovery for transaction systems interact in subtle ways. In most theoretical work, however, concurrency control and recovery are treated as separate, largely independent problems. In this paper we investigate the interactions between concurrency control and recovery. We consider two general recovery methods for abstract data types, update-in-place and deferred-update. While each requires operations to conflict if they do not commute, the two recovery methods require subtly different notions of commutativity. We give a precise characterization of the conflict relations that work with each recovery method, and show that each permits conflict relations that the other does not. Thus, the two recovery methods place incomparable constraints on concurrency control. Our analysis applies to arbitrary abstract data types, including those with operations that may be partial or non-deterministic.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Judy Little, Publications Coordinator	(617) 253-5894	

The Impact of Recovery on Concurrency Control

(Extended Abstract)

William E. Weihl

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
(617) 253-6030
E-mail: weihl@lcs.mit.edu

Abstract

It is widely recognized by practitioners that concurrency control and recovery for transaction systems interact in subtle ways. In most theoretical work, however, concurrency control and recovery are treated as separate, largely independent problems. In this paper we investigate the interactions between concurrency control and recovery. We consider two general recovery methods for abstract data types, update-in-place and deferred-update. While each requires operations to conflict if they do not "commute," the two recovery methods require subtly different notions of commutativity. We give a precise characterization of the conflict relations that work with each recovery method, and show that each permits conflict relations that the other does not. Thus, the two recovery methods place incomparable constraints on concurrency control. Our analysis applies to arbitrary abstract data types, including those with operations that may be partial or non-deterministic.

This research was supported in part by the National Science Foundation under Grant CCR-8716884 and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

1. Introduction

It is widely recognized by practitioners that concurrency control and recovery for transaction systems interact in subtle ways. In most theoretical work, however, concurrency control and recovery are treated as separate, largely independent problems. For example, most theoretical papers on concurrency control tend to ignore recovery, assuming that some unspecified recovery mechanism ensures that aborted transactions have no effect, and then considering only executions in which no transactions abort. In this paper we investigate the interactions between concurrency control and recovery. We show that the choice of recovery method constrains the possible choices for concurrency control algorithms, and that different recovery methods place incomparable constraints on concurrency control. Existing work on concurrency control is not invalidated by these results; rather, implicit assumptions about recovery in prior work are identified and made explicit.

Atomic transactions have been widely studied for over a decade as a mechanism for coping with concurrency and failures, particularly in distributed systems [12, 20, 7, 1]. A major area of research during this period has involved the design and analysis of concurrency control algorithms, for which an extensive theory has been developed (e.g., see [17, 3]). Initial work in the area left the data uninterpreted, or viewed operations as simple reads and writes. Recently, a number of researchers have considered placing more structure on the data accessed by transactions, and have shown how this structure can be used to permit more concurrency [9, 21, 23, 22, 18, 1, 2, 15, 25, 24, 16]. For example, in our own work we have shown how the specifications of abstract data types can be used to permit high levels of concurrency [21, 23], by designing type-specific concurrency control algorithms that take advantage of algebraic properties of a type's operations. Such techniques have been used in existing systems to deal with "hot-spots." In addition, such techniques are useful in general distributed systems, and may also prove useful in object-oriented database systems.

In contrast to the vast theoretical literature on concurrency control, there has been relatively little theoretical work on recovery, although some work does exist [8]. Hadzilacos analyzes several crash recovery methods, and addresses the question of what constraints are needed on concurrency control for the recovery methods to work. However, he assumes an update-in-place model for recovery, and analyzes only single-version read-write databases. In addition, the recovery methods studied by Hadzilacos, based on logging values, will not work with concurrency control algorithms that permit concurrent updates. More complex recovery algorithms, based on intentions lists or undo operations, have been designed for these more sophisticated concurrency control algorithms that permit concurrent updates. However, a theory of their interactions is sadly lacking.

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

We focus in this paper on recovery from transaction aborts, and ignore crash recovery. Crash recovery mechanisms are frequently similar to abort recovery mechanisms, but are also usually more complex due to the need to cope with the uncertainties about exactly what information might be lost in a crash. Thus, we expect a similar analysis to apply to many crash recovery mechanisms. To simplify the problem, however, we ignore crash recovery here, and leave its analysis for future work.

We consider two general recovery methods, update-in-place and deferred-update, and show that they place incomparable constraints on concurrency control. While each requires operations to conflict if they do not "commute," the two recovery methods require subtly different notions of commutativity. We give a precise

characterization of the conflict relations that work with each recovery method, and show that each permits conflict relations that the other does not. Our analysis applies to arbitrary abstract data types, including those with operations that may be partial or non-deterministic. In addition, our analysis covers concurrency control algorithms in which the lock required by an operation may be determined by the results returned by the operation, as well as by its name and arguments.

We use dynamic atomicity [21, 23] as our correctness criterion. Dynamic atomicity characterizes the behavior of many popular concurrency control algorithms, including most variations of two-phase locking [5, 9, 18, 22]. Dynamic atomicity is a local atomicity property, which means that if every object in a system is dynamic atomic, transactions will be atomic (i.e., serializable and recoverable).[1] This means that different concurrency control and recovery algorithms can be used at different objects in a system, and as long as each object is dynamic atomic, the overall system will be correct.

The remainder of this paper is organized as follows. In Section 2, we summarize our computational model, and in Section 3, we summarize the definitions of atomicity and dynamic atomicity. Then, in Section 4, we describe a high-level model for concurrency control and recovery algorithms that permits us to focus on their interactions while ignoring many implementation details. In Section 5, we describe the two recovery methods, and in Section 6, we define several different notions of commutativity. Next, in Section 7, we give a precise characterization of the conflict relations that work with each of the two recovery methods defined in Section 5. Finally, in Section 8, we conclude with a brief summary of our results.

2. Computational Model

Our model of computation is taken from [21, 23]; we summarize the relevant details here. There are two kinds of entities in our model: *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 will typically use the symbols A, B, and C for transactions, and X, Y, and Z for objects. We use ACT to denote the set of transactions.

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

- Invocation events, denoted <inv, X, A>, occurs when a transaction A invokes an operation of object X. The "inv" field includes both the name of the operation and its arguments.
- Response events, denoted <res, X, A>, occur when an object returns a response res to an earlier invocation by transaction A of an operation of object X.
- Commit events, denoted <commit, X, A>, occur when object X learns that transaction A has committed.
- Abort events, denoted <abort, X, A>, occur when object X learns that transaction A has aborted.

We say that event <e, X, A> *involves* X and A.

We introduce some notation here. If H is a history, let *Committed*(H) be the set of transactions that commit in H; similarly, define *Aborted*(H) to be the set of transactions that abort in H. Define *Active*(H) to be the set of active transactions in H; i.e., Active(H) = ACT – Committed(H) – Aborted(H). Also, if H is a history and X is a (set of) object(s), define H|X to be the subsequence of H consisting of the events involving (the objects in) X; similarly define H|A for a (set of) transaction(s) A.

---

[1] Dynamic atomicity is in fact an *optimal* local atomicity property: no strictly weaker property of individual objects suffices to ensure global atomicity of transactions.

A computation is modeled as a sequence of events. To simplify the model, we consider only finite sequences. The properties of interest in this paper are safety properties, and finite sequences suffice for analyzing such properties. Not all finite sequences, however, make sense. For example, a transaction should not commit at some objects and abort at others, and should not continue executing operations at objects after it has committed. To capture these constraints, we introduce a set of *well-formedness* constraints. A well-formed finite sequence of events is called a *history*. We summarize the well-formedness constraints here; details can be found in [21, 23]:

- Each transaction A must wait for the response to its last invocation before invoking the next operation, and an object can generate a response for A only if A has a pending invocation.
- Each transaction A can commit or abort in H, but not both; i.e., committed(H|A) $\cap$ aborted(H|A) = $\emptyset$.
- A transaction A cannot commit if it is waiting for the response to an invocation, and cannot invoke any operations after it commits.

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 [6, 11, 19].

We will typically use juxtaposition (e.g., $\alpha\beta$) to denote concatenation of sequences, but will use the symbol • to denote concatenation when juxtaposition is too hard to read. We use Λ to denote the empty sequence.

3. Atomicity

In this section we define atomicity and several related properties. Most of the definitions are abstracted from [21, 23]. In this abstract, we provide only brief informal definitions intended to convey the intuition needed to understand the later descriptions and discussion; complete formal details are in the full paper.

3.1. I/O Automata

I/O automata [13] are a convenient tool for describing concurrent and distributed systems. We will use I/O automata in several ways in this paper. For example, we will model an implementation of an object as an I/O automaton. We will also use I/O automata as a way of describing specifications of objects: we model a specification as a set of sequences (or traces), which is just a language, and an automaton is a convenient tool for describing a language.

We assume minimal familiarity with the details of I/O automata; we summarize the relevant details here. An I/O automaton consists of: a *state set*, a subset of which are designated as *initial states*; a set of *actions*, partitioned into *input* and *output* actions; and a *transition relation*, which is a set of triples of the form (s',π,s), where s' and s are states and π is an action.[2] The elements of the transition relation are called *steps* of the automaton.

If there exists a state s such that (s',π,s) is an element of the transition relation, we say that π is enabled in s'. An I/O automaton is required to be *input-enabled*: every input action must be enabled in every state.

A finite sequence $\alpha = \pi_1...\pi_n$ of actions is said to be a *schedule* of an I/O automaton if there exist states $s_0,...,s_n$ such that s_0 is a start state, and each triple (s_{i-1},π_i,s_i) is a step of the automaton for $1\le i\le n$. We define the *language*

[2] An I/O automaton can also have internal actions and an additional component characterizing the fair executions; we omit these here since we do not need them in the rest of the paper.

of an I/O automaton M, denoted L(M), to be the set of schedules of M.

3.2. Specifications

Each object has a *serial specification*, which defines its behavior in the absence of concurrency and failures, as well as a *behavioral specification*, which characterizes its behavior in the presence of concurrency and failures. The behavioral specification of an object X is simply a set of histories that contain only events involving X.

The serial specification of an object X, denoted Spec(X), is intended to capture the acceptable behavior of X in a sequential, failure-free environment. We could model the serial specification of X as a set of histories, where the histories satisfy certain restrictions (e.g., all transactions commit, and events of different transactions do not interleave). We have found it convenient, however, to use a slightly different model for serial specifications. Instead of a set of histories, we will use a prefix-closed set of *operation sequences*. (Prefix-closure means that if a sequence α is in the set, any prefix β of α is also in the set.) An *operation* is a pair consisting of an invocation and a response to that invocation; in addition, an operation identifies the object on which it is executed.

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

X:[insert(3),ok]

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

If an operation sequence α is in Spec(X), we say that α is *legal according to* Spec(X). If Spec(X) is clear from context, we will simply say that α is legal.

We will typically use I/O automata to describe serial specifications, by defining Spec(X) to be the language of some I/O automaton whose actions are the operations of X. For example, consider a bank account object BA, with operations to deposit and withdraw money, and to retrieve the current balance. Assume that a withdrawal has two possible results, "ok" and "no." Spec(BA) is the language of an I/O automaton M(BA), defined as follows. A state s of M(BA) is a non-negative integer; the initial state is 0. The output actions of M(BA) are the operations of BA; there are no input actions. The steps (s',π,s) of M(BA) are defined by the preconditions and effects below for each action π. We follow the convention that an omitted precondition is short for a precondition of true, and an omitted effects indicates that s=s'.

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

π = BA:[withdraw(i),ok], i > 0
Precondition:
s' ≥ i
Effects:
s = s' − i

π = BA:[withdraw(i),no], i > 0
Precondition:
s' < i

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

Spec(BA) includes the following sequence of operations:

BA:[deposit(5),ok]
BA:[withdraw(3),ok]
BA:[balance,2]
BA:[withdraw(3),no]

However, it does not include the following sequence:

BA:[deposit(5),ok]
BA:[withdraw(3),ok]
BA:[balance,2]
BA:[withdraw(3),ok]

The withdraw operation returns "ok" if and only if the current balance is not less than the argument of the operation; the first sequence above satisfies this constraint, while the second does not.

3.3. 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 do this by defining a function *Opseq* from histories to operation sequences. Opseq is defined inductively as follows. First, Opseq(Λ) = Λ. Second, Opseq(H•e), where e is a single event, is just Opseq(H) if e is an invocation, commit or abort event; if e is a response event <R,X,A>, and <I,X,A> is the pending invocation for A in H, then Opseq(H•e) = Opseq(H)•X:[I,R]. In other words, Opseq(H) is the operation sequence that contains the operations in H in the order in which they occur (i.e., the order of the response events); commit and abort events and pending invocations are ignored.

We say that a serial failure-free history H (one in which events for different transactions are not interleaved, and in which no transaction aborts) is *acceptable at X* if Opseq(H|X) is legal according to Spec(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.

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

If H is a failure-free history and T is a partial order on transactions that totally orders the transactions that appear in H, we then say that 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 an order T 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 bank account object BA is atomic:

<deposit(3), BA, A>

<ok, BA, A>
<withdraw(2), BA, B>
<ok, BA, B>
<balance, BA, A>
<3, BA, A>
<balance, BA, B>
<commit, BA, A>
<!, BA, B>
<commit, BA, B>
<withdraw(2), BA, C>
<no, BA, C>
<commit, BA, C>

The history contains only committed transactions, and is serializable in the order A followed by B followed by C.

3.4. 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 [21, 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 *dynamic atomicity*. Most concurrency control algorithms, including two-phase locking [5, 4, 9], determine a serialization order for transactions *dynamically*, based on the order in which transactions invoke operations and obtain locks on objects. Dynamic atomicity characterizes the behavior of algorithms that are dynamic in this sense. Informally stated, the fundamental property of protocols characterized by dynamic atomicity is the following: if the sequence of operations executed by one committed transaction conflicts with the operations executed by another committed transaction, then some of the operations executed by one of the transactions must occur after the other transaction has committed. In other words, if two transactions are completely concurrent at the object (neither executes an operation after the other commits), they must not conflict. Locking protocols (and all pessimistic protocols) achieve this property by *delaying* or *refusing* conflicting operations; optimistic protocols [10] achieve this property by allowing conflicts to occur, but *aborting* conflicting transactions when they try to commit to prevent conflicts among committed transactions.

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

The following lemma from [21, 23] provides the key to our definition of dynamic atomicity.

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

If H is a history of the system, each object has only partial information about precedes(H). However, if each object X ensures local serializability in *all* orders consistent with precedes(H|X), then by Lemma 1 we are guaranteed global serializability in all orders consistent with precedes(H). To be precise, we have the following definition of dynamic atomicity: we say that a history H is *dynamic atomic* if permanent(H) is serializable in every total order consistent with precedes(H). In other words, every serial history equivalent to permanent(H), with the transactions in an order consistent with precedes(H), must be acceptable.

The following theorem, taken from [21, 23], justifies our claim that dynamic atomicity is a local atomicity property:

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

As an example, the history H illustrated at the end of Section 3.3 is dynamic atomic as well as atomic: it is serializable in the order A-B-C, and since a response event for B occurs after the commit event for A, and similarly a response event for C occurs after the commit event for B, this is the only total order consistent with precedes(H). However, if the last response event for B occurred before the commit event for A, the history would not be dynamic atomic, since then (A,B) would not be in precedes(H), but the history is not serializable in the order B-A-C.

4. Concurrency Control and Recovery Algorithms

We adopt the following as the correctness criterion for an implementation of an object. First, we view an implementation of an object as an I/O automaton I whose actions are the events involving the object. Now, we say that I is *correct* if every history in L(I) is dynamic atomic. Our goal in this paper is to explore which combinations of concurrency control and recovery algorithms lead to correct implementations.

Different implementations of objects differ greatly in the details of the steps they perform to execute an operation invoked by a transaction. Viewed at a high level, an implementation might do the following:

1. Acquire any locks needed (waiting if there are conflicts).
2. Determine the "state" of the object.
3. Choose a result consistent with the state found in the previous step.
4. Update the state if necessary.
5. Record recovery data.
6. Return the result chosen in step 3.

Some implementations might execute these steps in a different order, or might use a completely different breakdown. For example, some implementations might use the result of an operation, as well as its name and arguments, to determine the locks required by the operation. Other implementations might allow several operations to run concurrently, relying on short-term locks (e.g., page locks) held for the duration of each operation to prevent them from interfering with each other.

Most of these differences among implementations are irrelevant as far as the interactions between concurrency control and recovery are concerned. To be reasonably general, and to avoid getting bogged down in complex implementation details, we adopt the following more abstract model of an object's implementation. We view an implementation of an object X as an I/O automaton I(X,Spec,View,Conflict). Spec is the serial specification of X, View is an abstraction of the recovery algorithm to be used, and Conflict is an abstraction of the concurrency control algorithm to be used. Spec is a set of operation sequences; the types of View and Conflict are defined more precisely below.

The actions of I(X,Spec,View,Conflict) are simply the events involving X. More precisely, the input actions of I(X,Spec,View,Conflict) are the invocation, commit, and abort events involving X; the outputs are the response

events involving X. Thus, the object receives invocations, commits, and aborts from transactions, and can generate responses to invocations.

We use perhaps the most abstract model possible for the states of I(X,Spec,View,Conflict): a state of I(X,Spec,View,Conflict) is simply a sequence of events. The initial state is the empty sequence. When an event involving the object takes place, it is appended to the state. Thus, the state of the object records the events involving the object in the order they happen. Of course, an actual implementation would use a much more efficient representation for the state of an object, but such implementation details are not relevant for our analysis.

The input events are always enabled, since they are controlled by the transactions. However, we will assume that transactions preserve the well-formedness constraints discussed earlier. Response events are enabled if there are no concurrency conflicts, and if the response being returned is consistent (according to the serial specification Spec(X)) with the current state of the object.

More precisely, let *Conflict* be a binary relation on operations. The relation Conflict is used by I(X,Spec,View,Conflict) to test for conflicts: a response <R,X,A> can occur for an invocation <I,X,A> only if the operation X:[I,R] does not conflict with any operation already executed by other active transactions. The conflict relation between operations is the essential variable in conflict-based locking.

Recovery is modelled by a function *View* from histories and active transactions to operation sequences. The function View can be thought of as defining the "serial state" (represented as an operation sequence) used to determine the legal responses to an invocation. View models recovery from aborts in the sense that the serial state used by an operation to determine its response should ignore the operations executed by aborted transactions. We will show in the next section how View can be used to model different recovery methods. First, however, we present the transitions of I(X,Spec,View,Conflict) more formally.

Formally, the transitions (s',π,s) of I(X,Spec,View,Conflict) are described by the preconditions and effects given below for each action π:

π is an invocation event <I,X,A>
Effects:
$s = s'\pi$

π is a response event <R,X,A>
Precondition:
A has a pending invocation I in s
$\forall$ transactions $B \in$ Active(s),
$\forall$ operations P in Opseq(s|B),
$(X:[I,R],P) \notin$ Conflict
View(s,A) • $X:[I,R] \in$ Spec(X).
Effects:
$s = s'\pi$

π is a commit event <commit,X,A>
Effects:
$s = s'\pi$

π is an abort event <abort,X,A>
Effects:
$s = s'\pi$

As stated above, each event is simply recorded in the state when it occurs. The first precondition for response events ensures that I(X,Spec,View,Conflict) preserves well-formedness: a response event is generated only for transactions with pending invocations. The second precondition tests whether the locks required by the operation

can be obtained. The locks acquired by a transaction are implicit in the operations it has executed; locks are released implicitly when a transaction commits or aborts (since then it is no longer active). The third precondition constrains the responses that can be generated: they must be legal according to Spec(X) after the operation sequence View(s,A).

An actual implementation could test the preconditions on response events in any order, and if there are several legal responses might always choose a particular one. Our model abstracts from such details. We note that not all algorithms can be modelled in this way. For example, the test for concurrency conflicts considered here is independent of the current state of the object. Nevertheless, many interesting algorithms, including most published type-specific concurrency control and recovery algorithms (e.g., [18, 22, 9, 2, 25]), fit into this framework. In the remainder of this paper we will explore constraints on Conflict and View that guarantee that I(X,Spec,View,Conflict) is correct. We will consider two different recovery methods, and show that they place incomparable constraints on conflict relations.

5. Recovery

In this section we present two different recovery methods, and show how to model them in terms of a View function. The first method is called "update in place" (or UIP). UIP is an abstraction of recovery algorithms in which a single "current" state is maintained. When a transaction executes an operation, the current state is used to determine the response to the operation, and is modified to reflect any changes (e.g., inserting a tuple) performed by the operation. When a transaction commits, nothing needs to be done, since the current state already reflects the effects of the transaction's operations. When a transaction aborts, however, the effects of the transaction's operations on the current state must be "undone" in some fashion. Most database systems, including System R [7], use an update-in-place strategy for recovery from transaction aborts.

The details of undoing operations can be complex. We abstract from them by defining the view based on the entire history. More precisely, we define the function UIP for a history H and a transaction $A \in Active(H)$ as follows: $UIP(H,A) = Opseq(H \mid ACT - Aborted(H))$. In other words, UIP computes a serial state by including all the operations executed by non-aborted transactions, in the order in which they were executed (i.e., the order in which their responses occurred).

The second recovery method is called "deferred update" (or DU). DU is an abstraction of recovery algorithms based on intentions lists, in which the base copy of the database is not updated until a transaction commits [11]. Alternatively, one can think of each transaction as having its own private workspace with a copy of the database in which it makes changes; these changes are not seen by other transactions until it commits. The way in which a transaction executes an operation depends on the implementation. If we use private workspaces, the state in the transaction's private workspace is used to determine the response to the operation, and the private workspace is updated to reflect any changes performed by the operation. If we use intentions lists, the base copy of the database is used to determine the response to the operation, except that the effects of the operations already in the transaction's intentions list must be accounted for; the intentions list is updated simply by appending the new operation. Aborts are simple for DU, since the intentions list or private workspace can just be discarded. Commits can be harder, depending on the implementation. If we use intentions lists, we simply have to apply the transaction's intentions list to the base copy of the database. If we use private workspaces, we have to update the base copy appropriately, but may also have to update the private workspaces of other active transactions to ensure that the effects of committed transactions are made visible to active transactions. Relatively few systems seem to use a deferred-update strategy for recovery from transaction aborts, perhaps because executing an operation and committing a transaction can be more expensive than when an update-in-place strategy is used. Nevertheless, this strategy has been used in some systems, notably XDFS and CFS [14].

More precisely, we define the function DU for a history H and a transaction A∈ Active(H) as follows. First, define the total order *Commit-order*(H) on transactions that commit in H to contain exactly those pairs (A,B) such that the first commit event for A occurs in H before the first commit event for B.[3] Now, define DU(H,A) = Opseq(Serial(H|Committed(H),Commit-order(H))) • Opseq(H|A). In other words, DU computes a serial state by including all the operations of committed transactions, in the order in which they committed, followed by the operations already executed by the transaction A itself.

DU and UIP both include the effects of the operations of committed transactions, and of the particular active transaction A. They differ in the order of these operations: UIP includes them in the order in which the operations occurred, while DU includes them in the order in which the transactions committed, followed by A. They also differ in whether the effects of other active transactions are included: UIP includes the effects of *all* non-aborted transactions, both committed and active, while DU includes the effects of only the committed transactions and the particular active transaction A.

A simple example serves to illustrate the differences between DU and UIP. Consider the following history H involving a bank account object BA:

<deposit(5),BA,A>
<ok,BA,A>
<commit,BA,A>
<withdraw(3),BA,B>
<ok,BA,B>

UIP(H,B) is the following operation sequence (corresponding to an account balance of 2):

BA:[deposit(5),ok]
BA:[withdraw(3),ok]

Since UIP gives the same result regardless of the transaction, UIP(H,C), for some other transaction C, is the same operation sequence. Since B is the only active transaction in H, DU(H,B) is also the same operation sequence. However, DU(H,C) is the sequence

BA:[deposit(5),ok]

which contains only the operations executed by the committed transactions.

One might think that these rather subtle differences between DU and UIP are irrelevant. Indeed, much of the literature on concurrency control seems to be based on the implicit assumption that concurrency control and recovery can be studied independently, and that different recovery methods such as DU and UIP can all be regarded as implementations of some more abstract notion of recovery. For example, recovery is typically handled by assuming that there is some recovery method that ensures that aborted transactions "have no effect," and then considering only executions in which no transactions abort when analyzing concurrency control. In the process, however, most people seem to assume a model for recovery similar to UIP. As we will show, this assumption is non-trivial: DU and UIP work correctly with different — in fact, incomparable — classes of concurrency control algorithms.

We note that many other View functions are possible. We have begun by studying UIP and DU because they are abstractions of the two most common recovery methods in use. One interesting question for future work is whether there are other View functions that place weaker constraints on concurrency control than UIP or DU.

[3]Commit-order(H) is defined for all histories H; however, we will make use of the definition only for histories involving a single object. The same is true of UIP and DU.

6. Commutativity

Each of the recovery methods described in the previous section works in combination with a conflict relation based on "commutativity:" two operations conflict if they do not "commute." However, the different recovery algorithms require subtly different notions of commutativity. In this section we describe the two definitions and give some examples to illustrate how they differ.

It is important to point out that we define the two notions of commutativity as binary relations on operations in the sense of our formal definition, rather than simply for invocations as is usually done. Thus, the locks acquired by an operation can depend on the results returned by the operation. In addition, it is convenient to phrase our definitions in terms of sequences of operations, not just individual operations.

6.1. Equieffectiveness

To define commutativity, it is important to know when two operation sequences lead to the same "state." Rather than defining commutativity in terms of the "states" of objects, however, we take a more abstract view based on the sequence of operations applied to an object.

First, if Spec is a set of operation sequences and α and β are operation sequences, we say that α *looks like* β *with respect to* Spec if for every operation sequence γ, $\alpha\gamma \in$ Spec only if $\beta\gamma \in$ Spec. In other words, α looks like β if, after executing α, we will never see a result of an operation that allows us to distinguish β from α. Notice that the relation "looks like" is not necessarily symmetric (although it is reflexive and transitive).

Second, if Spec is a set of operation sequences and α and β are operation sequences, we say that α and β are *equieffective with respect to* Spec (or that α is equieffective to β with respect to Spec) if α looks like β with respect to Spec and β looks like α with respect to Spec. In other words, α and β are indistinguishable by future operations.

We include here some simple properties of these definitions.

Lemma 3: The relation "looks like with respect to Spec" is reflexive and transitive.

Lemma 4: The relation "equieffective with respect to Spec" is an equivalence relation.

Lemma 5: If $\alpha \in$ Spec and α looks like β or α is equieffective to β with respect to Spec, then $\beta \in$ Spec.

Lemma 6: If α looks like β with respect to Spec, then $\alpha\gamma$ looks like $\beta\gamma$ with respect to Spec for all γ.

Lemma 7: If α and β are equieffective with respect to Spec, then $\alpha\gamma$ and $\beta\gamma$ are equieffective with respect to Spec for all γ.

6.2. Forward Commutativity

If Spec is a set of operation sequences, and β and γ are operation sequences, we say that β and γ *commute forward with respect to* Spec if, for every operation sequence α such that $\alpha\beta \in$ Spec and $\alpha\gamma \in$ Spec, $\alpha\beta\gamma$ is equieffective to $\alpha\gamma\beta$ with respect to Spec and $\alpha\beta\gamma \in$ Spec. The motivation for the terminology is that whenever β and γ can each be executed after some sequence α, each can be pushed forward past the other.

Define the relation FC(Spec) to be the binary relation on operations containing all pairs (β,γ) such that β and γ commute forward with respect to Spec. Define the relation NFC(Spec) to be the complement of FC(Spec).

Lemma 8: FC(Spec) and NFC(Spec) are symmetric relations.

For example, the forward commutativity relation on operations of the bank account object BA is given by the table in Figure 6-1. Deposits and successful withdrawals do not commute with balance operations, since the former change the state. Similarly, successful withdrawals do not commute with each other; for example, each of

BA:[withdraw(i),OK] and BA:[withdraw(j),OK] is legal after any operation sequence α that results in a net balance greater than or equal to max(i,j), but if the net balance after α is less than i+j then the two withdrawal operations cannot be executed in sequence after α.

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

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

Figure 6-1: Forward Commutativity Relation for BA

6.3. Backward Commutativity

If Spec is a set of operation sequences, and β and γ are operation sequences, we say that β *right commutes backward with* γ *with respect to* Spec if, for every operation sequence α, $\alpha\gamma\beta$ looks like $\alpha\beta\gamma$ with respect to Spec. The motivation for the terminology is that whenever β can be executed immediately after (i.e., to the right of) γ, it can be pushed backward so that it is before γ.

Define the relation RBC(Spec) to be the binary relation on operations containing all pairs (β,γ) such that β right commutes backward with γ with respect to Spec. Define the relation NRBC(Spec) to be the complement of RBC(Spec).

Notice that RBC(Spec) and NRBC(Spec) are not necessarily symmetric. Most previous work (including some of our own) assumes, sometimes implicitly, that conflict relations must be symmetric. We will show that UIP works with *Conflict* if and only if NRBC(Spec)$\subseteq$*Conflict*. If we required conflict relations to be symmetric, we would be forced to include additional conflicts that are not necessary. (In particular, *Conflict* would have to contain the symmetric closure of NRBC(Spec).)

The right backward commutativity relation for the bank account object BA is described in Figure 6-2. For example, suppose P=BA:[withdraw(j),OK] and Q=BA:[deposit(i),ok], let α be such that $\alpha QP \in$ Spec(BA), and let s' be the state of M(BA) after α. Then by the precondition for P, $s'+i \geq j$, so $s' \geq j-i$. If α is such that $s' < j$, then $\alpha PQ \notin$ Spec(BA), so P does not right commute backward with Q. However, Q does right commute backward with P: if the withdrawal (P) can be executed before the deposit (Q), it can also be executed after the deposit since the deposit increases the balance (and the two sequences are equieffective since addition commutes).

6.4. Discussion

The rather subtle differences between the two notions of commutativity are shown by comparing Figure 6-2 to Figure 6-1: the forward and right backward commutativity relations are incomparable. We will show that UIP works in combination with exactly those conflict relations that contain NRBC(Spec), while DU works in combination with exactly those conflict relations that contain NFC(Spec). Since in general NRBC(Spec) and NFC(Spec) are incomparable, this implies that these two recovery methods place incomparable constraints on concurrency control.

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

*× indicates that the operation for the given row does **not** right commute backward with the operation for the column.*

Figure 6-2: Right Backward Commutativity Relation for BA

7. Interaction of Recovery and Concurrency Control

In this section we characterize the conflict relations that work with UIP and with DU. The proofs make use of the following additional definitions. First, if H is a history and CS is a set of transactions, we say that CS is a *commit set* for H if committed(H) $\subseteq$ CS and CS $\cap$ aborted(H) = $\emptyset$. In other words, CS is a set of transactions that have already committed or might commit. Second, we say that H is *online dynamic atomic* if, for every commit set CS for H, H|CS is serializable in every total order consistent with precedes(H|CS). It is immediate that H is dynamic atomic if it is online dynamic atomic.

The conflict relations that work with an update-in-place recovery method are characterized by the following theorem:

Theorem 9: I(X,Spec,UIP,Conflict) is correct if and only if NRBC(Spec)$\subseteq$Conflict.

Proof: For the if direction, suppose NRBC(Spec)$\subseteq$Conflict, and let H be a history in L(I(X,Spec,UIP,Conflict)). We show that H is online dynamic atomic, which implies that H is dynamic atomic. The proof is by induction on the length of H. If H = Λ, the result is immediate. Otherwise, suppose H = K•<e,X,A>, and let CS be a commit set for H. By induction, K is online dynamic atomic. There are now two cases. First, if e is an invocation of an operation, e=commit, e=abort, or A$\notin$ CS, then CS is also a commit set for K, and Opseq(H|CS) = Opseq(K|CS) and precedes(H|CS) = precedes(K|CS), so the result holds by induction.

Second, suppose e is the response R to an invocation I, and A$\in$ CS. Let Q be the operation X:[I,R]. By the precondition for e, UIP(H,A)•Q is legal. We need to show that Serial(H|CS,T) is legal for every T consistent with precedes(H|CS). Let α = Serial(H|CS$\cup$Active(H),T'), where T' is consistent with T on CS but orders the elements of Active(H)–CS after the elements of CS. Serial(H|CS,T) is a prefix of α. Since Spec(X) is prefix-closed, it suffices to show that α is legal. It is easy to show that there is a sequence UIP(H,A)•Q = α_0, α_1, ..., α_n = α of operation sequences such that α_i can be obtained from α_{i-1} by swapping two adjacent operations — i.e., α_{i-1} = βPQγ, and α_i = βQPγ — and (Q,P)$\notin$ Conflict. Since NRBC(Spec)$\subseteq$Conflict, (Q,P)$\in$ RBC(Spec), so α_{i-1} looks like α_i. By Lemma 3, UIP(H,A)•Q looks like α. By Lemma 5, α is legal.

For the only if direction, suppose (P,Q)$\in$ NRBC(Spec) but (P,Q)$\notin$ Conflict. We show that there is a history H in L(I(X,Spec,UIP,Conflict)) that is not dynamic atomic. Since (P,Q)$\in$ NRBC(Spec), there exists an α such that αQP does not look like αPQ with respect to Spec. Then there must be some ρ such that αQP$\rho\in$ Spec but αPQ$\rho\notin$ Spec. Let H be the history constructed as follows:

A executes the operation sequence α at X
A commits at X
B executes Q at X
C executes P at X

B commits at X
C commits at X
D executes the operation sequence ρ at X
D commits at X

H is permitted by I(X,Spec,UIP,Conflict). However, it is not dynamic atomic, since neither B nor C precedes the other, yet it is not serializable in the order A-C-B-D (because $\alpha PQ\rho \notin$ Spec). □

The conflict relations that work with a deferred-update recovery method are characterized by the following theorem:

Theorem 10: I(X,Spec,DU,Conflict) is correct if and only if NFC(Spec)$\subseteq$Conflict.

Proof: The if direction is a relatively straightforward induction; proofs can be found in [21, 22]. For the only if direction, suppose (P,Q)$\in$ NFC(Spec) but (P,Q)$\notin$ Conflict. We show that there is a history H in L(I(X,Spec,DU,Conflict)) that is not dynamic atomic. Since (P,Q)$\in$ NFC(Spec), there exists an α such that $\alpha P \in$ Spec and $\alpha Q \in$ Spec, and either $\alpha PQ \notin$ Spec or αPQ is not equieffective to αQP with respect to Spec.

There are two cases. First, if $\alpha PQ \notin$ Spec, let H be the history constructed as follows.

A executes the operation sequence α at X
A commits at X
B executes P at X
C executes Q at X
B commits at X
C commits at X

H is permitted by I(X,Spec,DU,Conflict). However, it is not dynamic atomic. Since neither B nor C precedes the other, dynamic atomicity requires that H be serializable in the orders A-B-C and A-C-B. But $\alpha PQ \notin$ Spec, so H is not serializable in the order A-B-C.

Second, suppose αPQ is not equieffective to αQP with respect to Spec. Then there is some ρ such that either $\alpha PQ\rho \in$ Spec and $\alpha QP\rho \notin$ Spec, or $\alpha QP\rho \in$ Spec and $\alpha PQ\rho \notin$ Spec. Without loss of generality, suppose $\alpha PQ\rho \in$ Spec and $\alpha QP\rho \notin$ Spec. Let H be the history constructed as follows:

A executes the operation sequence α at X
A commits at X
B executes P at X
C executes Q at X
B commits at X
C commits at X
D executes the operation sequence ρ at X
D commits at X

H is permitted by I(X,Spec,DU,Conflict). However, it is not dynamic atomic, since neither B nor C precedes the other, yet it is not serializable in the order A-C-B-D (because $\alpha QP\rho \notin$ Spec). □

8. Conclusions

We have analyzed two general recovery methods for abstract data types, and have given necessary and sufficient conditions for conflict relations to work with each. The classes of conflict relations that work in combination with the two recovery methods are incomparable, implying that choosing between these two recovery methods involves a trade-off in concurrency: each permits conflict relations that the other does not, and thus there may be applications for which one or the other is preferable on the basis of the level of concurrency achieved.

Most of the concurrency control literature assumes an update-in-place model for recovery. The results in this paper show that this is not just a technical assumption: other recovery methods permit conflict relations not permitted by update-in-place, and thus cannot be viewed simply as implementations of update-in-place.

Most, if not all, approaches to recovery of which we are aware can be viewed as implementations of either update-in-place or deferred-update, so we believe that the analysis of these specific methods is quite important. However, there are interesting algorithms that combine aspects of both of these two methods, or in which concurrency control and recovery are more tightly integrated. For example, O'Neil has presented a type-specific concurrency control and recovery algorithm in which concurrency control and recovery are tightly coupled, and in which the test for conflicts depends on the current state of the object [16]. Further work is required to characterize the recovery algorithms that can be modeled using the framework presented in this paper, and to generalize our approach to accomodate other algorithms. It would be interesting to consider concurrency control algorithms other than the conflict-based locking algorithms considered here, and to consider correctness conditions other than dynamic atomicity.

The material presented here grew out of earlier work [22], in which we presented two locking algorithms for abstract data types. One of the two algorithms in [22] is essentially a combination of DU with a conflict relation of NFC; the other is a combination of UIP with a more restrictive conflict relation than NRBC. In [22] we proved the correctness of the two algorithms, and conjectured that it was impossible to do better for either recovery method. In addition, the model of an implementation used in [22] is relatively low-level, containing many details that turn out not to be important; the model of an implementation presented in this paper more clearly highlights the interactions between concurrency control and recovery. The results here provide a precise characterization of the conflict relations that work with each recovery method, confirming our earlier conjecture for DU but disproving it for UIP. The algorithm consisting of the combination of UIP and NRBC(Spec) presented in this paper is interesting in itself, since it requires fewer conflicts than previous algorithms, and also because it is impossible to do better.

9. References

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

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

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

[4] Bernstein, P., Goodman, N., and Lai, M.-Y.
Analyzing concurrency control when user and system operations differ.
IEEE Transactions on Software Engineering SE-9(3):223-239, May, 1983.

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

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

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

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

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

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

[11] 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.

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

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

[14] Mitchell, J. G. and Dion, J.
A comparison of two network-based file servers.
Communications of the ACM 25(4):233-245, April, 1982.
Special issue: Selected papers from the Eighth Symposium on Operating Systems Principles.

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

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

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

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

[21] 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.

[22] W.E. Weihl.
Commutativity-based Concurrency Control for Abstract Data Types.
IEEE Transactions on Computers 37(12):1488-1505, December, 1988.
Also available as MIT/LCS/TM-367.

[23] W.E. Weihl.
Local atomicity properties: modular concurrency control for abstract data types.
ACM Transactions on Programming Languages and Systems 11, 1989.
Accepted for publication.

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

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

OFFICIAL DISTRIBUTION LIST

Director
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

2 copies

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

2 copies

Director, Code 2627
Naval Research Laboratory
Washington, DC 20375

6 copies

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

12 copies

National Science Foundation
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

2 copies

Dr. E.B. Royce, Code 38
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

1 copy