TIMESTAMP ORDERING AND NESTED TRANSACTIONS

by

James D. Aspnes

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
ENGINEERING AND COMPUTER SCIENCE IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1987

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
May 8, 1987

Certified by ______________________________
Nancy Lynch
Thesis Supervisor

Accepted by ______________________________
A. C. Smith
Chairman, Committee on Graduate Students

TIMESTAMP ORDERING AND NESTED TRANSACTIONS

by

James D. Aspnes

Submitted to the Department of Electrical Engineering and Computer
Science on May 8, 1987 in partial fulfillment of the requirements for the
degree of Master of Science in Electrical Engineering and Computer
Science.

Abstract

Using the [LM] model for database concurrency control and recovery, we describe a general method for proving serial correctness of concurrency control algorithms which use timestamp ordering. This method is then used to prove serial correctness of Reed's [R] object history mechanism.

Thesis Supervisor: Nancy Lynch
Title: Professor of Computer Science, Massachusetts Institute of Technology

Table of Contents

Chapter 1

Introduction

Much work in the theory of database concurrency control has focused on the question of whether particular algorithms guarantee *serializability*, in the sense that some (usually external) observer cannot distinguish between the effects of executions in which transactions are run concurrently or serially. The notion of serializability has recently been generalized [LM]to produce a correctness condition for systems of nested transactions; in this model, a particular execution of a system is said to be *serially correct* if, for each transaction, the execution "looks like" an execution of a serial system.

In this paper we describe a general method for proving serial correctness for executions of nested transaction systems which use timestamp ordering algorithms for concurrency control. Timestamp ordering algorithms are interesting in that they require the system to construct explicitly the order in which transactions could appear in a serial execution. The existence of the explicit ordering allows more concurrency than is usually possible with locking algorithms; in particular, it allows a transaction to commit at any time, regardless of the state of its subtransactions. On the other hand, the presence of this additional concurrency means that we need to construct a sophisticated theoretical framework before we can begin to prove that it does not violate correctness.

Some of the work of building this framework has already been done, in [LM]. We present several significant additions to this work which are of particular use in proving serial correctness for timestamp ordering algorithms, but which should also be useful in proving properties of other concurrency control algorithms. Principle among these is an *affects ordering* based on causal dependencies in the serial system; from this ordering we obtain both a mechanism for determining precisely what part of any execution of a system can be detected by a particular transaction, and a simple method of testing an arbitrary ordering on transactions to determine if it yields a reordering of a particular execution which is consistent with the behavior of the serial system.

The organization of the paper is as follows. Chapter 2 reviews those parts of the model described in [LM] and [FLMW] which are relevant to nested transaction systems using timestamp ordering. Chapter 3 describes a general method for proving serial correctness of timestamp ordering algorithms, independent of the mechanism used to generate and communicate the timestamp order; we close the chapter with a statement and proof of a theorem which clearly defines the essential properties of any correct timestamp ordering algorithm. Finally, Chapter 4 proves rigorously, for the first time, that the object history mechanism of Reed [R] guarantees serial correctness.

Chapter 2

The Model

In this chapter, we will describe the basic concepts and definitions which will be necessary for our discussion of timestamp-ordered systems. The first three sections are mostly a restatement of material found in [LM] and [FLMW]. The last sections describe some terminology which will we need to describe the effects of timestamp ordering.

2.1 I/O Automata

We represent the components of a nested transaction system by I/O automata. An I/O automaton A consists of five components: states(A), start(A), out(A), in(A), and steps(A). In this model, states(A) is a set of states, of which a subset start(A) are designated as start states. In(A) and out(A) represent the set of input and output operations, respectively; we require these sets to be disjoint, and refer to their union as the set of *operations* of the automaton. Steps(A) is the transition relation of A, consisting of triples of the form (s',π,s), where s' and s are states, and π is an operation. Each triple (s',π,s), called a *step* of A, means that A, when in state s', may atomically perform the operation π and change to state s. π is said to be *enabled* in s' if (s',π,s) is in steps(A) for some s.

Output operations are intended to represent those actions which the automaton triggers itself; the input operations represent those which are triggered by the automaton's environment. We require that an I/O automaton be able to receive any input at any time, which we express formally as the following condition.

Input Condition:

For each input operation π and state s', π is enabled in s.

An *execution* of A is a finite alternating sequence $s_0,\pi_1,s_1,\ldots,s_n$ of states and operations of *A*, where s_0 is a start state of *A*, and each consecutive subsequence (s',π,s) is in steps(*A*). From any execution, we may extract a *schedule*, which is the subsequence of the execution which contains only the operations. Since transitions to different states may have the same operation, different executions may have the same schedule. For the purposes of this paper, we will assume that all schedules are finite.

If P is a property of schedules, then *A* is said to *preserve* P if, for any $\alpha = \alpha'\pi$ which is a schedule of *A* where α' has property P and π is an output operation, then α has property P.

We describe systems as collections of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata as well. Thus we define a composition operation for I/O automata.

A set of I/O automata may be composed to create a *system S*, provided that the sets of output operations of the automata are disjoint. The states of *S* are tuples of states, one for each component; the start states of *S* are those tuples consisting of start states of the components. The set of operations of *S*, ops(*S*), is the union of the sets of operations of the component automata. Similarly the set of output operations, out(*S*), is the union of the sets of output operations of the components. The set of input operations, in(*S*), is simply ops(*S*) - out(*S*). As with a single automaton, the output operations represent actions which are triggered by some part of the system, and the input operations represent actions which are triggered externally.

The triple (s',π,s) is in the transaction relation of *S* if and only if for each component automaton *A*, one of the following two conditions holds. Either π is an operation of *A*, and the projection of the step onto *A* is a step of *A*, or else π is not an operation of *A*, and the states corresponding to *A* in the two tuples s' and s are identical. Thus each operation of the composed automaton is an operation of a subset of the component automata. During an operation π of *S*, each of the components which has operation π carries out the operation, while the remainder stay in the same state.

An execution of a system is defined to be an execution of the automaton composed of the individual automata of the system. If α is a sequence of operations of a system with component A, then the projection of α on A, $\alpha|A$, is the subsequence of α containing only the operations of A. Clearly, if α is a schedule of S, $\alpha|A$ is a schedule of A.

The following lemma expresses formally the principle that an operation of the system is under the control of the component of which it is an output. The proof is given in [LM].

Lemma 2.1: Let α' be a schedule of a system S, and let $\alpha = \alpha'\pi$, where π is an output operation of component A. If $\alpha|A$ is a schedule of A, then α is a schedule of S.

2.2 Serial Systems

In this section, we define *serial systems*, which consist of transactions, basic objects, and a serial controller. Transactions and basic objects represent user programs and data, respectively; the controller controls all communication among the other components, and thereby restricts the orders in which transactions may execute. Each component of the serial system is modeled as an I/O automaton.

The nesting of transactions is specified by a *system type*. A system type is a four-tuple $<T,\text{parent},O,V>$. T is the set of transaction names; parent is a mapping from T to T which organizes T into a tree. In referring to this tree, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor, and descendant. (A transaction is its own ancestor and descendant). The leaves of T are called *accesses*. The set O is a partition of the set of accesses, where each element of O contains the accesses to a particular object. For convenience, we will denote the objects of the system by the elements of O corresponding to them. Lastly, V is the set of *values* that transactions may return.

We assume that the transaction tree is known in advance to all components of the system. T can be thought of as a predefined naming scheme for all transactions which might ever be executed. It will, in general, be infinite, but only finitely many transactions will take steps in any particular execution.

The classical transactions of concurrency control (without nesting) appear in this model as the children of a "mythical" transaction T_0, the root of the transaction tree. T_0 itself represents the environment in which the rest of the transaction system runs, and has operations which represent the invocation and return of the classical transactions. In practice, we will not distinguish between T_0, its children, and other internal nodes of *T*.

2.2.1 Transactions

We consider it important not to place unnecessary constraints on the behavior of transactions. Thus, rather than require that this behavior be expressible in a particular high-level programming language, we represent transactions as (possibly infinite-state) I/O automata. This approach allows us to state precisely those properties that are relevant to the behavior of transactions as part of a transaction system, while avoiding any restrictions on, or need to describe, the actual computations being performed by the transactions.

We model each non-access transaction T as an I/O automaton with the following operations:

Input:
  CREATE(T)
  REPORT-COMMIT(T',v), for each T' $\in$ children(T) and v $\in$ *V*
  REPORT-ABORT(T'), for each T' $\in$ children(T)
Output:
  REQUEST-CREATE(T'), for each T' $\in$ children(T)
  REQUEST-COMMIT(T,v), for each v $\in$ *V*

The CREATE(T) operation "wakes up" the transaction. The REPORT-COMMIT and REPORT-ABORT operations, which we refer to together as *reports*, are the mechanism by which the controller informs transactions of the state of their offspring. The REQUEST-CREATE operation requests the creation of a particular child transaction; it is distinct from the corresponding CREATE to allow the controller to delay or even deny the request. Similarly, the REQUEST-COMMIT operation only requests that the transaction be committed, with the actual COMMIT of the transaction being controlled by the controller.

For the most part, we do not specify the executions of particular transaction automata. The choices of which children to create, when to create them, and when to request to return are left up to the particular implementation of the transaction. We will be concerned only that the transaction preserve well-formedness, which we define recursively as follows.

Definition 2.2: Let T be a transaction, and let α be a schedule of T. If α is the empty sequence, α is well-formed. Alternatively, let $\alpha = \alpha'\pi$, where π is a single operation. Then α is well-formed provided α' is well-formed, and the following hold:

- If π is CREATE(T), then
  1. CREATE(T) does not appear in α'.
- If π is REPORT-COMMIT(T',v) for a child T' of T, then
  1. REQUEST-CREATE(T') appears in α',
  2. REPORT-ABORT(T') does not appear in α', and
  3. REPORT-COMMIT(T',v') does not appear in α' for any v' $\neq$ v.
- If π is REPORT-ABORT(T') for a child T' of T, then
  1. REQUEST-CREATE(T') appears in α', and
  2. no REPORT-COMMIT for T appears in α'.
- If π is REQUEST-CREATE(T') for a child T' of T, then
  1. REQUEST-CREATE(T') does not appear in α',
  2. REQUEST-COMMIT(T) does not appear in α', and
  3. CREATE(T) appears in α'.
- If π is REQUEST-COMMIT(T,v), then
  1. there is no REQUEST-COMMIT for T in α', and
  2. CREATE(T) appears in α'.

These restrictions are very basic. They say simply that a transaction is never created more than once, does not receive reports from children it has not requested, makes no requests more than once, and takes no action before it is created or after it requests to commit. Except for these minimal conditions, we place no restrictions on the behavior of transactions. For example, a transaction may request to commit without yet knowing the status of subtransactions whose creation it has requested, and may request to create new subtransactions without regard to its state of knowledge about subtransactions whose creation it has previously requested. It may be that a particular programming language or system might place additional restrictions on the behavior of transactions. However, our results do not require such restrictions.

The following easy lemma summarizes some properties of well-formed sequences of transaction operations.

Lemma 2.3: Let α be a well-formed sequence of operations of transaction T. Then the following conditions hold.

1. The first operation of α is a CREATE(T) operation, and there are no other CREATE operations in α.
2. If a REQUEST-COMMIT operation occurs in α, then there are no later output operations in α.
3. α contains at most one REQUEST-CREATE(T') operation for each child T' of T.
4. For every report operation in α, there is an earlier REQUEST-CREATE operation in α for the same child transaction.

2.2.2 Basic Objects

We use basic object automata to represent user data. Recall that each basic object X is associated with a set of access transactions which is an element of O. We refer to this set as accesses(X); these transactions correspond to the operations provided for manipulating and examining a particular basic object. As with transaction automata, we leave much of the specific behavior of basic objects unspecified. The limited well-formedness conditions we place on the behavior of basic automata will stem primarily from our desire that access transactions satisfy the same well-formedness conditions as non-access transactions.

A basic object X has the following operations:

Input:
CREATE(T), for each T $\in$ accesses(X).
Output:
REQUEST-COMMIT(T,v), for each T $\in$ accesses(X) and v $\in$ V.

We will abuse our notation by letting $\alpha|T$ refer to the subsequence of α consisting only of operations of T, even when T is a non-access transaction. We may then use the following simple definition for a well-formed sequence of basic object operations.

Definition 2.4: Let α be a sequence of operations of a basic object X. Then α is well-formed if and only if $\alpha|T$ is a well-formed sequence of operations of T for each T in accesses(X).

The following lemma describes well-formed sequences of basic object operations in a slightly more convenient form.

Lemma 2.5: Let α be a sequence of operations of a basic object X. If α is the empty sequence, then α is well-formed. Otherwise let $\alpha = \alpha'\pi$; α is well-formed if α' is well-formed and the following conditions hold:

- If π is CREATE(T), then
  1. CREATE(T) does not appear in α'.
- If π is REQUEST-COMMIT(T,v), then
  1. CREATE(T) appears in α', and
  2. REQUEST-COMMIT(T,v') does not appear in α', for any value v'.

Proof: Immediate from Definitions 2.2 and 2.4.

We require that each basic object preserve well-formedness.

2.2.3 Serial Controller

The third component of the serial system is the serial controller. Unlike the transactions and basic objects, the serial controller is a fully-specified automaton. It runs transactions according to a depth-first traversal of the transaction tree, and has the power to abort any transaction whose creation has been requested, as long as it has not already been created. The serial controller also waits for children to return before allowing their parent to commit. A formal description of the serial controller, adapted from [LM, FLMW], follows.

The serial controller has the following operations, for each $T \in T$ and $v \in V$:

Input:
REQUEST-CREATE(T), $T \neq T_0$
REQUEST-COMMIT(T,v)
Output:
CREATE(T)
ABORT(T), $T \neq T_0$
COMMIT(T,v), $T \neq T_0$
REPORT-ABORT(T), $T \neq T_0$

REPORT-COMMIT(T,v), $T \neq T_0$

Each serial controller state s has components s.create-requested, s.created, s.aborted, s.commit-requested, and s.committed. The s.create-requested component lists those transactions for which the controller has received a REQUEST-CREATE; s.created stores the names of all transactions which have actually been created. Similarly, s.committed and s.aborted are the set of all transactions which have committed or aborted, respectively. The component s.commit-requested is a set of <transaction,value> pairs; it records all REQUEST-COMMITs. In the initial state of the controller, s.created = $\{T_0\}$, and all of the other components are empty.

Although it is not an actual component of the controller state, we will write s.returned as shorthand for s.committed $\cup$ s.aborted. The steps of the controller will be exactly those triples (s',π,s) satisfying the following pre- and postconditions. Note that not all of the components of s are specified in the postconditions for each operation. Unspecified components are assumed not to change between s' and s.

- REQUEST-CREATE(T)
  Postcondition:
  s.create-requested = s'.create-requested $\cup$ {T}
- REQUEST-COMMIT(T,v)
  Postcondition:
  s.commit-requested = s'.commit-requested $\cup$ {(T,v)}
- CREATE(T)
  Preconditions:
  T $\in$ s'.create-requested
  T $\notin$ s'.created $\cup$ s'.aborted
  siblings(T) $\cap$ s'.created $\subseteq$ s'.returned
  Postcondition:
  s.created = s'.created $\cup$ {T}
- ABORT(T)
  Precondition:
  T $\in$ s'.create-requested
  T $\notin$ s'.created $\cup$ s'.aborted
  Postconditions:
  s.aborted = s'.aborted $\cup$ {T}
- REPORT-ABORT(T)

Precondition:
$T \in s'.\text{aborted}$

- COMMIT(T,v)
  Preconditions:
  $(T,v) \in s'.\text{commit-requested}$
  $T \notin s'.\text{returned}$
  $\text{children}(T) \cap s'.\text{create-requested} \subseteq s'.\text{returned}$
  Postcondition:
  $s.\text{committed} = s'.\text{committed} \cup \{T\}$
- REPORT-COMMIT(T,v)
  Preconditions:
  $(T,v) \in s'.\text{commit-requested}$
  $T \in s'.\text{committed}$

Some details of this controller are worth noting. The preconditions on CREATE(T) guarantee that no transaction will be created until all of its siblings that have already been created have returned. This condition yields a depth-first traversal of the transaction tree, which we claim is the natural notion of serial execution in a nested transaction system.

We have separated both the act of creating a transaction and the act of committing a transaction into several operations. This separation prevents a parent transaction from specifying or detecting the order in which its children are run unless it waits for a report from each before starting the next, or the children interact by accessing the same basic object. While the flexibility we have given the controller is not really necessary in the serial system, it will become important when we use the serial system as the basis for a correctness condition for more sophisticated systems.

A pleasant feature of the serial controller is the simplicity of its postconditions. It is not difficult to reconstruct the state of the controller at the end of an execution from the schedule of that execution.

Lemma 2.6: Let α be a schedule of the serial controller which can lead to a state s from the initial state. Then all of the following are true:

1. $T \in s.\text{create-requested}$ if and only if $T = T_0$ or REQUEST-CREATE(T) appears in α.
2. $(T,v) \in s.\text{commit-requested}$ if and only if REQUEST-COMMIT(T,v) appears in α.

3. T $\in$ s.created if and only if CREATE(T) appears in α.
4. T $\in$ s.aborted if and only if ABORT(T) appears in α.
5. T $\in$ s.committed if and only if COMMIT(T,v) appears in α for some v.

Proof: By induction on α using the controller postconditions.

2.2.4 Serial Schedules

The serial system is the composition of the transactions, basic objects, and the serial controller. In this section we present some terminology which will be useful for talking about the serial system.

We refer to the operations of the serial system as *serial operations*. Similarly, a *serial schedule* is a schedule of the serial system. For an arbitrary sequence of operations α, serial(α) is that subsequence of α consisting of only the serial operations.

We call the transactions and basic objects of the serial system the system *primitives*. A sequence of serial operations is well-formed if its projection on each of the system primitives is well-formed. A proof of the well-formedness of serial schedules is given in [FLMW]; we will not reproduce it here.

The operations ABORT(T) and COMMIT(T,v), for all v $\in$ V, are the *return* operations for T. Similarly, REPORT-ABORT(T) and REPORT-COMMIT(T,v) constitute the *report* operations for T. A serial operation ϕ *mentions* T if ϕ is an operation of T, or ϕ is a return operation for T; for example, CREATE(T), REQUEST-COMMIT(T,v), and COMMIT(T,v) all mention T, while REQUEST-CREATE(T) and REPORT-COMMIT(T,v) mention parent(T). Every serial operation mentions some transaction.

If α is an arbitrary sequence of operations, and T a transaction, we say T is *committed* in α if α contains a COMMIT for T; we say T is *aborted* in α if α contains an ABORT for T. T is an *orphan* in α if any ancestor of T is aborted in α.

2.3 Correctness Condition

The serial system has the advantage of simplicity. Transactions are run sequentially, and are atomic in the sense that aborted transactions were never created, and thus can have no effect on the system, and committed transactions must have run to completion. Unfortunately, the serial controller's simplicity carries a price of inefficiency. It is impossible to run transactions concurrently, even when they do not affect each other in any way, and it is impossible for the system to abort transactions once are running. We would like, then, to be able to build a more capable system, which would nonetheless appear to retain the simplicity of the serial system.

It is not immediately clear what conditions we would need to place on a more powerful system to preserve the appearance of the serial system. One possibility is to define a notion of database consistency, and require that the data in the system satisfy consistency at specified points in its execution.[1] This approach has the drawback of constraining ways in which the database can be represented in the system, and of reducing the applicability of results obtained in one system to other systems with radically different mechanisms for storing data. We believe that a better condition is that given in [LM], which requires only that no transaction can detect that the system is not serial.

> **Definition 2.7:** Let α be an arbitrary sequence of operations, some of which may be serial. Then α is *serially correct* for a primitive P if its projection on P is identical to the projection on P of some serial schedule. We say that a sequence of operations is *serially correct* if it is serially correct for each non-access transaction.

From the point of view of a systems implementor, serial correctness has several desirable properties. As it depends only on the projection of a schedule on non-access transactions, it places no restraints on the interface to or nature of the objects of the system. Thus the correctness condition is applicable, without modification, to algorithms which use

[1] An example of the use of a consistency predicate can be found in [BHG].

multiple copies of objects, which provide additional information to objects, or which share the functions of a basic object across several automata, so long as the interface to transactions is maintained in the same form as in the serial system. In fact, because the correctness condition allows different transactions to see different serial schedules, the correctness condition allows us to consider systems in which it is impossible to construct a meaningful global state. Thus serial correctness gives great leeway to a system and its designer.

On the other hand, serial correctness is not as weak a condition as it may seem. If all schedules of a system are serial correct, no transaction can tell that the system is not the serial system. But the set of transactions includes the mythical transaction T_0, which represents the outside environment of the system. So serial correctness guarantees the appearance of serial execution not only to the transactions, but to the outside world as well.

It is often convenient to further weaken serial correctness. For example, some systems may have schedules which are serially correct only for T_0 or for transactions whose ancestors have not been aborted. We will demonstrate that Reed's algorithm satisfies the latter condition (and thus the former, since T_0 has no ancestors besides itself, and cannot be aborted). In fact, it is possible to make slight modifications to any system which satisfies the weaker condition to produce a system which is serially correct for all transactions; see [HLMW].

2.4 Events

For the concurrent and weak concurrent systems of [LM], it is possible to construct a serial schedule from a concurrent schedule by taking a subsequence. For timestamp-ordered systems, however, it is in general necessary to reorder a schedule in order to obtain a serial schedule, as will become clear later. Thus we will need to be able to specify orderings on operations.

Unfortunately, it is possible for a schedule of even the serial controller to contain duplicate operations, and we will need to be a little careful in defining our orderings. We refer to a specific instance of an operation in a sequence of operations as an *event*. We will not define events formally; it is sufficient for our purposes to assume that all instances of an operation in any sequence of operations are distinct events, and our use of the term will not be sophisticated enough to require any explicit naming scheme. We denote the set of events occurring in a particular sequence of operations α as the *events of* α.

We will abuse our terminology somewhat by referring to an event in terms of the operation of which it is an instance. So, for example, by a serial event we will mean an event which is an instance of a serial operation, and by an event of T, where T is a transaction, we will mean an event which is an instance of an operation of T.

We now define some terminology which will be useful in discussing orderings on events. Let E be a binary relation on events. Then, if α is a sequence of operations, we say α is E-*ordered* provided E partially orders the events of α,[2] and for any pair of events (ϕ,π) in E, ϕ precedes π in α. A sequence β is a *reordering* of α if the set of events of β is equal to the set of events of α. If E partially orders the events of α, we denote by reorder(α,E) an arbitrary E-ordered reordering of α. We say a subsequence β of α is E-*closed* if, for any event π in β and ϕ in α such that $(\phi,\pi) \in$ E, ϕ is in β. If β is an arbitrary subsequence of α, we denote the smallest E-closed subsequence of α containing β by closure(β, α, E).

Note that in the above discussion we have not required E to be a relation solely on the events of α. Occasionally it will be useful to restrict E to a particular set of events. Accordingly, if S is a set of events, we write $E|S$ for the restriction of E to S. We will also write $E|\alpha$, where α is a sequence of operations, for the restriction of E to the events of α.

[2] By which we mean E is a strict partial order when restricted to the events of α; we will assume throughout that all partial orders are strict, i.e. irreflexive.

2.5 Sibling Orders

The essential feature of timestamp ordering algorithms is the explicit definition of an ordering on transactions which corresponds to the order of execution in the serial system. In the most general case we will specify this ordering by a sibling order, as defined below.

Definition 2.8: Let SIB be the set $\{(T_1,T_2) \mid T_1 \neq T_2, T_1$ is a sibling of $T_2\}$. Then $R \subseteq$ SIB is a *sibling order* just in case R is a partial order. R is a *total sibling order* if, for any (T_1,T_2) in SIB, either (T_1,T_2) or (T_2,T_1) is in R.

If R is a sibling order, we define R^*, the *descendant closure* of R, to be the set $\{<T_1,T_2> \mid$ there exists $<U_1,U_2> \in R$, $T_1 \in$ descendants(U_1), $T_2 \in$ descendants$(U_2)\}$ Clearly, R^* is a partial order on the set of transactions. The descendant closure of a total sibling order has some useful properties.

Lemma 2.9: Let R be a total sibling order. Then if T, T' are transactions neither of which is an ancestor of the other, either (T,T') or (T',T) is in R^*.

Proof: If neither T nor T' is an ancestor of the other, there must exist distinct ancestors U and U' of T and T' which are children of lca(T,T'). Since R is a total sibling order, either (U,U') or (U',U) is in R; thus either (T,T') or (T',T) is in R^*.

If R is an arbitrary binary relation on transactions, we can define a corresponding relation R_E on events.

Definition 2.10: Let R be a binary relation on transactions. Then we define the relation R_E to be the set $\{<\phi,\pi> \mid \phi$ mentions T_1, π mentions T_2, and $<T_1,T_2> \in R\}$.

If R is a partial order on transactions, it is not difficult to see that R_E must be a partial order on events. When R is a sibling order, we will primarily be interested in R^*_E, the event order defined by the descendant closure of R.

Chapter 3

Timestamp Ordering in Generic Systems

In this chapter, we will define a generic system, and describe conditions under which a general form of timestamp ordering can yield serial correctness in this system.

3.1 Generic Systems

The generic system consists of a generic controller, generic objects, and the transactions of the serial system.

3.1.1 Generic Controller

The generic controller has the following operations, for each object X, transaction T, and value v:

Input:
REQUEST-CREATE(T), $T \neq T_0$
REQUEST-COMMIT(T,v)
Output:
CREATE(T)
ABORT(T), $T \neq T_0$
COMMIT(T,v), $T \neq T_0$
REPORT-ABORT(T), $T \neq T_0$
REPORT-COMMIT(T,v), $T \neq T_0$
INFORM-ABORT-AT(X)OF(T), $T \neq T_0$
INFORM-COMMIT-AT(X)OF(T), $T \neq T_0$

These operations include all of the operations of the serial controller, and add only the INFORM-ABORT and INFORM-COMMIT operations. States of the generic controller have the same components as states of the serial controller, and the initial state is also the same.

The steps of the generic controller are exactly those transitions (s',π,s) satisfying the following pre- and postconditions:

- REQUEST-CREATE(T)
  Postcondition:
  s.create-requested = s'.create-requested $\cup$ {T}
- REQUEST-COMMIT(T,v)
  Postcondition:
  s.commit-requested = s'.commit-requested $\cup$ {(T,v)}
- CREATE(T)
  Preconditions:
  T $\in$ s'.create-requested - s'.created
  Postcondition:
  s.created = s'.created $\cup$ {T}
- ABORT(T)
  Precondition:
  T $\in$ s'.create-requested - s'.returned
  Postconditions:
  s.aborted = s'.aborted $\cup$ {T}
- REPORT-ABORT(T)
  Precondition:
  T $\in$ s'.aborted
- COMMIT(T,v)
  Preconditions:
  (T,v) $\in$ s'.commit-requested
  T $\notin$ s'.returned
  Postcondition:
  s.committed = s'.committed $\cup$ {T}
- REPORT-COMMIT(T,v)
  Preconditions:
  (T,v) $\in$ s'.committed-requested
  T $\in$ s'.committed
- INFORM-ABORT-AT(X)OF(T)
  Precondition:
  T $\in$ s'.aborted
- INFORM-COMMIT-AT(X)OF(T)
  Precondition:
  T $\in$ s'.committed

As with the serial controller, the state of the generic controller following a particular schedule can be easily deduced.

Lemma 3.1: Let α be a schedule of the generic schedule that can lead to a state s from the initial state. Then all of the following are true:

1. T $\in$ s.create-requested if and only if T = T_0 or REQUEST-CREATE(T) appears in α.

2. (T,v) ∈ s.commit-requested if and only if REQUEST-COMMIT(T,v) appears in α.
3. T ∈ s.created if and only if CREATE(T) appears in α.
4. T ∈ s.aborted if and only if ABORT(T) appears in α.
5. T ∈ s.committed if and only if COMMIT(T,v) appears in α for some v.

Proof: By induction on α using the controller postconditions.

The following lemma describes some of the properties of schedules of the generic controller.

Lemma 3.2: Let α be a schedule of the generic controller. Then all of the following hold:

1. If a CREATE(T) event appears in α, a REQUEST-CREATE(T) event precedes it in α.
2. If a COMMIT(T,v) event appears in α, a REQUEST-COMMIT(T,v) event precedes it in α.
3. If an ABORT(T) event appears in α, a REQUEST-CREATE(T) event precedes it in α.
4. If a REPORT-COMMIT(T,v) event or an INFORM-COMMIT-AT(X)OF(T) event appears in α, a COMMIT(T,v) event precedes it in α.
5. If a REPORT-ABORT(T) event or an INFORM-ABORT-AT(X)OF(T) event appears in α, an ABORT(T) event precedes it in α.
6. At most one CREATE event appears in α for each transaction.
7. At most one return event appears in α for each transaction.

Proof: By induction on α using Lemma 3.1 and the controller preconditions.

As can be seen by the preceding lemma, the generic controller embodies those constraints that we would expect any reasonable controller to satisfy. Thus the generic controller does not allow CREATEs, ABORTs, or COMMITs without an appropriate preceding request; does not report returns that never happened; and does not allow any transaction to return more than once. On the other hand, the generic controller allows almost any ordering of transactions, and allows arbitrary concurrency. In fact, our generic controller is even more flexible than the generic controllers of [FLMW] or [HLMW], in that it allows parents to return before their children do.

We do not claim that our generic controller produces serially correct schedules in all executions; instead we use the generic controller as a base on which to build more sophisticated controllers, such as the pseudotime controller of Chapter , which implement specific timestamp ordering algorithms. The generic controller provides us with a means of describing the common features of timestamp ordering algorithms without requiring us to commit ourselves to a particular method of constructing or using timestamps.

3.1.2 Generic Objects

Each basic object X in the serial system is represented in the generic system by a *generic object* G(X). G(X) has the following operations:

Input:
CREATE(T), for each T in accesses(X)
INFORM-ABORT-AT(X)OF(T), for each $T \neq T_0$
INFORM-COMMIT-AT(X)OF(T), for each $T \neq T_0$
Output:
REQUEST-COMMIT(T,v), for each T in accesses(X) and each v in *V*

The new operations INFORM-ABORT and INFORM-COMMIT are the same as the new operations of the generic controller, and are intended to allow the generic object to use additional knowledge about the state of the transactions of the system to behave in a manner which allows serial correctness even if transactions are executing concurrently, or are aborted after creation.

Well-formedness for sequences of generic object operations is slightly more complicated than for basic objects. We define well-formedness recursively, as follows:

Definition 3.3: Let α be a sequence of operations of the generic object G(X). Then α is well-formed if α is the empty sequence, or if $\alpha = \alpha'\pi$, where π is a single operation, and the following conditions hold.

1. If π is CREATE(T), for T in accesses(X), then
   a. CREATE(T) does not appear in α'.
2. If π is REQUEST-COMMIT(T,v), then
   a. CREATE(T) appears in α', and
   b. REQUEST-COMMIT(T,v') does not appear in α' for any value of v'.

3. If π is INFORM-ABORT-AT(X)OF(T), then
   a. INFORM-COMMIT-AT(X)OF(T) does not appear in α'.

4. If π is INFORM-COMMIT-AT(X)OF(T), then
   a. INFORM-ABORT-AT(X)OF(T) does not appear in α', and
   b. if T $\in$ accesses(X), REQUEST-COMMIT(T,v) appears in α' for some v.

Lemma 3.4: Let α be a well-formed sequence of operations of a generic object G(X). Then serial(α) is a well-formed sequence of operations of X.

Generic objects are required to preserve well-formedness.

A generic object's view of the system is necessarily rather limited. It has the advantage over basic objects of being able to receive INFORM-COMMIT and INFORM-ABORT operations; unfortunately, the controller is not required ever to send these operations, so the knowledge of the system so received will often be incomplete. Nonetheless, the generic object must make do with what information is available to it. We will define some terms to describe the state of a generic object's knowledge of the system; these will be useful later, when we actually construct a generic object.

Let G(X) be a generic object and let α be a sequence of operations of G(X). If T is an access of X and T' a proper ancestor of T, we say that T is *committed at X* to T' in α if, for every U which is an ancestor of T and a proper descendant of T', α contains an instance of INFORM-COMMIT-AT(X)OF(U).[3] If T' is any transaction, then T is *visible at X* to T' in α if T is committed at X to lca(T,T') in α. We write $\text{visible}_X(\alpha,T')$ for the subsequence of serial(α) consisting only of operations whose transactions are visible at X to T'; it is not difficult to see that, when α is well-formed, $\text{visible}_X(\alpha,T')$ is a well-formed sequence of operations of the basic object X. Finally, we say that a transaction T is an *orphan at X* in α if, for some ancestor T' of T, α contains INFORM-ABORT-AT(X)OF(T').

In many schedules α of G(X), there will be accesses which cannot be visible to any

[3] This definition differs slightly from a similar definition in [FLMW], in that we do not require the INFORM-COMMIT-AT operations to appear in any particular order.

other transaction which is not an orphan at X, either because they are orphans at X in α, or because no REQUEST-COMMIT appears for them in α and thus no INFORM-COMMIT for them can appear without violating well-formedness. If we remove all of the operations of such accesses from serial(α), we obtain a sequence which will prove useful in defining correctness conditions for generic objects, in that it extracts that part of serial(α) which can have any effect on non-orphaned transactions.

Definition 3.5: Let α be a sequence of operations of a generic object G(X). Let done(α) be the subsequence of serial(α) consisting of the operations of all accesses T to G(X) which meet the following two criteria:

1. A REQUEST-COMMIT for T appears in α.
2. T is not an orphan at X.

If α is a well-formed sequence of operations of G(X), then done(α) will be a well-formed sequence of operations of X.

Lemma 3.6: Let α be a well-formed sequence of operations of G(X). Then done(α) is a well-formed sequence of operations of X.

Proof: Suppose otherwise. Then there is some access T such that done(α)|T is not well-formed. Now, done(α)|T = serial(α)|T, and serial(α) is well-formed by Lemma 3.4. But then done(α)|T is well-formed, by Definition 2.4. Thus done(α) is well-formed.

We will use done to define a condition on schedules of G(X) which describes whether they can be reordered according to a particular sibling order R to produce schedules of X. First we define the effect of R on a schedule of G(X). If α is a well-formed sequence of operations of G(X) and R a binary relation on transactions which totally orders the transactions mentioned in α, we write rearrange(α,R) for the unique well-formed R_E-ordered reordering of done(α).

We would like rearrange(α,R) to be a schedule of X. What we will actually require is somewhat stronger. Because done(α) uses only the information that is available in α, it is possible that G(X) might yet receive INFORM-ABORT operations which would alter done(α). Accordingly, we define our condition on schedules of G(X) in such a way as to allow G(X) to receive an INFORM-ABORT for any transaction for which no INFORM-COMMIT appears in α.

Definition 3.7: Let α be a schedule of a generic object G(X), and let R be a binary relation on transactions which totally orders the transactions mentioned in α. Then R *timestamp-orders* α if, for any sequence of INFORM-ABORT-AT(X) operations β such that αβ is well-formed, rearrange(αβ,R) is a schedule of the basic object X.

The following rather trivial lemma will be useful later on.

Lemma 3.8: Let α be a well-formed sequence of operations of a generic object G(X), and let T be any access of G(X). Let R be a binary relation on transactions which totally orders the accesses of G(X). Then if done(α)|T is nonempty, done(α)|T = rearrange(α,R)|T
= CREATE(T)REQUEST-COMMIT(T,v) for some value v.

Proof: If done(α)|T is nonempty, then α must contain REQUEST-COMMIT(T,v) for some value v. Then by well-formedness a CREATE(T) must precede this REQUEST-COMMIT(T,v) in α, and no other operations of T may appear in α. Thus done(α)|T = CREATE(T)REQUEST-COMMIT(T,v). Now, rearrange(α,R) is a well-formed reordering of done(α); thus it must contain both a CREATE(T) event and a REQUEST-COMMIT(T,v) event, in the same order as in done(α). Thus done(α)|T = rearrange(α,R)|T.

3.1.3 Generic Schedules

The generic system is the composition of the transactions, generic objects, and the generic controller. As with the serial system, we define a *generic schedule* to be a schedule of the generic system. *Generic operations* are operations of the generic system. If α is an arbitrary sequence of operations, generic(α) is defined to be the subsequence of α consisting only of the generic operations.

A generic schedule is well-formed if its projection on each transaction and generic object is well-formed. We show in the following lemma that all generic schedules are well-formed.

Lemma 3.9: Let α be a generic schedule. Then α is well-formed.

Proof: If α is the empty sequence, then α is trivially well-formed. Otherwise, let α = α'π, and suppose that α is well-formed. If π is an output operation of a transaction or a generic object, then α is well-formed by the requirements that transactions and generic objects preserve well-formedness. If π is not an output of a transaction or generic object, π must be an output of the generic controller, and one of the following conditions must hold.

- π is CREATE(T), where T is a non-access transaction. Then by Lemma 3.2, CREATE(T) cannot appear in α'. Thus α is well-formed.

- π is CREATE(T), where T is an access transaction. Again by Lemma 3.2, CREATE(T) cannot appear in α'. Thus α is well-formed.
- π is ABORT(T) or COMMIT(T,v). Then π is not an operation of any transaction or generic object, and α is well-formed.
- π is REPORT-ABORT(T). Then by Lemma 3.2, an ABORT(T) event occurs in α'. But then by the same lemma REQUEST-CREATE(T) must appear in α'. Now suppose that REPORT-COMMIT(T,v) appears in α' for some v; then COMMIT(T,v) also appears in α'. But then there would be more than one return for T in α', which contradicts Lemma 3.2. Thus α is well-formed.
- π is REPORT-COMMIT(T,v). Then COMMIT(T,v) appears in α', so REQUEST-COMMIT(T,v) appears in α', and (by well-formedness of α'), CREATE(T) appears in α'. Hence REQUEST-CREATE(T) must appear in α'. Now suppose some different report for T appears in α'; then some return for α' other than COMMIT(T,v) appears in α', which contradicts Lemma 3.2. Thus α is well-formed.
- π is INFORM-ABORT-AT(X)OF(T). Then ABORT(T) appears in α'. Suppose that INFORM-COMMIT-AT(X)OF(T) appears in α'; then some COMMIT for T appears in α'. But then there is more than one return for T in α', which contradicts Lemma 3.2. Thus INFORM-COMMIT-AT(X)OF(T) does not appear in α', and α is well-formed.
- π is INFORM-COMMIT-AT(X)OF(T). By an argument similar to that used for INFORM-ABORT-AT(X)OF(T), INFORM-ABORT-AT(X)OF(T) cannot occur in α'. There are now two cases, depending on whether or not T is an access of X. If it is not, then α is well-formed. Otherwise, we note that COMMIT(T,v) appears in α' for some v; thus REQUEST-COMMIT(T,v) must appear in α'. Thus α is well-formed.

3.2 The Affects Ordering

In order to construct a serial schedule for a particular transaction from a generic schedule using a sibling order, we will need first to remove events which cannot be detected by the given transaction, and then to reorder the resulting subsequence by R_E^*. For the first step, we will need some mechanism to determine which events to remove; for the second, we will need some finer ordering than R_E^* to preserve the ordering of events within transactions, and to order events of transactions which are not comparable by R^*. We can accomplish both tasks using the *affects* ordering, which we now define.

Definition 3.10: Let α be a sequence of serial operations, and let ϕ and π

be events of α. Then ϕ *directly affects* π in α (or, equivalently, $(\phi,\pi) \in$ directly-affects(α)) if one of the following conditions holds for some transaction T:

- ϕ and π are both events of T and ϕ precedes π in α.
- ϕ is an instance of REQUEST-CREATE(T) and π is an instance of CREATE(T).
- ϕ is an instance of REQUEST-CREATE(T) and π is an instance of ABORT(T).
- ϕ is an instance of REQUEST-COMMIT(T,v), and π is an instance of COMMIT(T,v).
- ϕ is a return event for T, and π is a COMMIT event for parent(T).
- ϕ is an instance of COMMIT(T,v) and π is an instance of REPORT-COMMIT(T,v).
- ϕ is an instance of ABORT(T) and π is an instance of REPORT-ABORT(T).

Affects(α) is the transitive closure of directly-affects(α). We will often say ϕ *affects* π *in* α for $(\phi,\pi) \in$ affects(α).

Our definition is by cases, as that form is the easiest for us to actually use. This choice of definition should not, however, obscure the underlying rationale behind the structure of affects(α). Affects(α) is intended to capture a basic property of the serial system, which is that the appearance of certain events in a serial schedule necessitates the prior appearance of certain other events. This requirement stems partly from the algorithm specified for the serial controller, partly from well-formedness, and partly from the behavior of transactions in the system. The first case of definition 3.10 stems from our assumption that we have no special knowledge of the possible schedules of any transaction; thus our only indication that a schedule of T is possible is that it is a prefix of $\alpha|T$. The other cases can easily be seen to follow from our construction of the serial controller.

We have not yet demonstrated that, when α is a well-formed sequence of serial operations, affects(α) is a partial order. First we will prove two lemmas which describe the relationship between affects(α) and the creation and return of transactions.

Lemma 3.11: Let α be a sequence of serial operations and T a transaction. Let ϕ and π be events of α such that ϕ mentions a descendant of T and π does not. Then if ϕ affects π in α, either ϕ is a return event for T, or ϕ affects a return event for T which affects π.

Proof: By definition of affects(α), if ϕ affects π in α there must exist a sequence $\phi,\phi_1,\phi_2,...,\pi$ in which each adjoining pair is in directly-affects(α). Since ϕ mentions a descendant of T and π does not, there must be some first pair ϕ_n, ϕ_{n+1} where ϕ_n mentions a descendant of T and ϕ_{n+1} does not. We note from Definition 3.10 that ϕ_n must be a return event for T, and ϕ_{n+1} is either a corresponding report event or a COMMIT event for parent(T). Either $\phi_n = \phi$, in which case ϕ is a return event for T; or ϕ affects ϕ_n, which affects π.

Lemma 3.12: Let α be a sequence of serial operations and T a transaction. Let ϕ and π be events of α such that ϕ does not mention a descendant of T and π does. Then if ϕ affects π in α, either ϕ is an instance of REQUEST-CREATE(T), or ϕ affects an instance of REQUEST-CREATE(T) in α.

Proof: By definition of affects(α), if ϕ affects π in α there must exist a sequence $\phi,\phi_1,\phi_2,...,\pi$ in which each adjoining pair is in directly-affects(α). As ϕ does not mention a descendant of T and π does, there must be some first pair ϕ_n, ϕ_{n+1} where ϕ_n does not mention a descendant of T and ϕ_{n+1} does. We note from Definition 3.10 that ϕ_n must be an instance of REQUEST-CREATE(T), and ϕ_{n+1} must be an instance of CREATE(T). Either $\phi_n = \phi$, in which case ϕ is an instance of REQUEST-CREATE(T); or ϕ affects ϕ_n, which affects π.

We may now show that affects(α) is a partial order on the events of α whenever α is well-formed.

Lemma 3.13: Let α be a well-formed sequence of serial operations. Then affects(α) is a partial order on the events of α.

Proof: Suppose otherwise. We know that affects(α) must be transitive, as it is a transitive closure; thus either it is not irreflexive or it is not antisymmetric. If it is not antisymmetric, there exists a pair (ϕ,π) in affects(α) such that (π,ϕ) is also in affects(α). But then $(\pi,\pi) \in$ affects(α) by transitivity, and affects(α) is not irreflexive. Thus if affects(α) is not a partial order, it is not irreflexive.

Suppose, then, that there exists some event ϕ in α which affects itself in α. Then ϕ mentions some transaction. Let T be a transaction such that some event of T in α affects itself in α, and no event of any proper ancestor of T affects itself in α. Let ϕ be the latest event mentioning T in α which affects itself in α. We note that, as affects(α) is the transitive closure of directly-affects(α), which is irreflexive, there must then exist some event π in α such that $(\phi,\pi) \in$ directly-affects(α) and $(\pi,\phi) \in$ affects(α); furthermore, π must also affect itself in α. We now consider the possible values of π.

If ϕ is a return event, then π must either be a report event of parent(T) or a return event for parent(T); in either case π mentions parent(T), which contradicts our choice of T.

Alternatively, if ϕ is a CREATE, REPORT-ABORT, REPORT-COMMIT, or REQUEST-COMMIT event, then π is either a later event of T or a COMMIT event for T. If π is a later event of T, ϕ is not the latest event mentioning T which affects itself, contrary to choice. If π is a COMMIT event for T, then π directly-affects only events mentioning parent(T); thus there is some event ρ mentioning parent(T) which affects itself, contradicting our choice of T.

Lastly, if ϕ is an instance of REQUEST-CREATE(T') for some child T' of T, then π is either a later event of T, an instance of ABORT(T'), or an instance of CREATE(T'). The first case contradicts our choice of ϕ as latest. In the second case, π must precede ϕ in α, so an instance of ABORT(T') precedes an instance of REQUEST-CREATE(T') in α, and α would not be well-formed, a contradiction. In the third case, π mentions a descendant of T' and ϕ does not; so by Lemma 3.11, π affects a COMMIT event ρ for T' and ρ affects ϕ. Now, ρ does not directly-affect ϕ, so ρ must directly-affect some event σ which affects ϕ in α. By examination of Definition 3.10, σ can be either a COMMIT for T or a REPORT-COMMIT for T'. We have already shown that no COMMIT event for T can affect itself in α. If σ is a REPORT-COMMIT for T', well-formedness requires that it follow ϕ; but then ϕ would not be the latest event mentioning T which affects itself in α, contrary to choice.

Thus affects(α) is irreflexive; it is therefore also antisymmetric, and, being transitive, must be a partial order.

The following lemma will be useful later, when we derive a sibling order from affects.

Lemma 3.14: Let α be a well-formed sequence of serial operations. Then if ϕ is a return event for T, and π is an instance of REQUEST-CREATE(T), ϕ does not affect π in α.

Proof: If ϕ is an ABORT, then π affects ϕ in α; thus if ϕ also affects π, affects(α) is not antisymmetric, contradicting Lemma 3.13.

Alternatively, ϕ must be a COMMIT for T. We assume without loss of generality that there is no ancestor U of T such that a COMMIT for U affects an instance of REQUEST-CREATE(U). Now, ϕ directly-affects only COMMIT events for parent(T) and REPORT-COMMIT events for T. Suppose that there is a COMMIT event ρ for parent(T) which affects π. Then ρ directly-affects only COMMIT events for parent(parent(T)) and REPORT-COMMIT events for parent(T). None of these events mention a descendant of parent(T), and π does, so if ρ affects π, ρ affects an instance of REQUEST-CREATE(parent(T)) by Lemma 3.12, contradicting our choice of T.

Alternatively, suppose there is a REPORT-COMMIT σ for T which affects π. Then σ must precede π in α, as both are events of parent(T). But then α would not be well-formed. Thus ϕ cannot affect π in α.

3.2.1 Affects and Sibling Orders

When we actually construct a serial schedule from a generic schedule, we will need to combine the affects ordering with an appropriate sibling order. Unfortunately, there will be some sibling orders which cannot be combined with affects without producing cycles. Fortunately, we can easily characterize the conditions which must be met by a sibling order in order for it to be compatible with affects. First, we define what we mean for two partial orders to be consistent.

Definition 3.15: Two partial orders R and S are *consistent* if the transitive closure of $R \cup S$ is a partial order.

We define $\text{affects}_T(\alpha)$ to be the binary relation on transactions derived from affects(α) by the following rule: for any pair of distinct sibling transactions (T_1,T_2), $(T_1,T_2) \in \text{affects}_T(\alpha)$ if and only if there is a pair of events (ϕ,π) in affects(α) such that ϕ mentions a descendant of T_1, and π mentions a descendant of T_2.

When α is well-formed, there is a simple method for extracting a superset of $\text{affects}_T(\alpha)$ from affects(α).

Lemma 3.16: Let α be a well-formed sequence of serial operations. Then if $(T_1,T_2) \in \text{affects}_T(\alpha)$, there is a report event for T_1 which affects an instance of REQUEST-CREATE(T_2) in α.

Proof: Let $(T_1,T_2) \in$ affects(α). Then there exists a pair of events $(\phi,\pi) \in$ affects(α) such that ϕ mentions a descendant of T_1 and π mentions a descendant of T_2. By Lemma 3.11, either ϕ is a return event for T_1 or ϕ affects a return event for T_1, ρ, which affects π in α. Now ρ directly affects only COMMIT events for parent(T_1) and report events for T_1. In the first case, if σ is a COMMIT event for parent(T) which affects π, then by Lemma 3.12 σ affects an instance of REQUEST-CREATE(parent(T_1)), contradicting Lemma 3.14.

Thus the alternative must hold, and there is a report event ψ for T_1 in α which affects π. ψ does not mention a descendant of T_2, and π does; thus by Lemma 3.12, ψ affects an instance of REQUEST-CREATE(T_2) in α.

Lemma 3.16 will be mostly useful later, when we wish to describe $\text{affects}_T(\alpha)$ for specific schedules α. It does, however allow us to prove that $\text{affects}_T(\alpha)$ is a sibling order when α is well-formed.

Lemma 3.17: Let α be a well-formed sequence of serial operations. Then $\text{affects}_T(\alpha)$ is a sibling order.

Proof: Clearly, $\text{affects}_T(\alpha) \subseteq$ SIB. We therefore need only prove that it is a partial order. Because it is a subset of SIB, non-siblings are incomparable; thus if we can prove $\text{affects}_T(\alpha)$ partially orders any set of siblings we will have proven that it partially orders all of *T*.

That $\text{affects}_T(\alpha)$ is irreflexive is immediate from the definition. To prove antisymmetry, suppose there exist two distinct siblings T_1, T_2 such that both (T_1,T_2) and (T_2,T_1) are in $\text{affects}_T(\alpha)$. Then by Lemma 3.16 there exists a REPORT event ϕ for T_1 which affects a REQUEST-CREATE event π for T_2, and a REPORT event ρ for T_2 which affects a REQUEST-CREATE event σ for T_1. All four of these events are events of the common parent of T_1 and T_2, so by well-formedness of α π must affect ρ and σ must affect ϕ. But then π,ρ,σ,ϕ,π is a cycle in affects(α), which contradicts Lemma 3.13.

Finally we must prove transitivity. Suppose (T_1,T_2) and (T_2,T_3) are both in affects$_T(\alpha)$. Then there exist events ϕ, π, and ρ of α mentioning descendants of T_1, T_2, and T_3, respectively, such that (ϕ,π) and (π,ρ) are both in affects(α). But then (ϕ,ρ) is in affects(α), and thus (T_1,T_3) is in affects$_T(\alpha)$ provided $T_1 \neq T_3$. But if $T_1 = T_3$, then affects$_T(\alpha)$ is not antisymmetric, a contradiction. Thus affects$_T(\alpha)$ is a partial order, and thus a sibling order.

The following lemma shows that, when α is a well-formed sequence of serial operations, and R a sibling order, it is sufficient for affects$_T(\alpha)$ to be consistent with R for affects(α) to be consistent with R_E^*.

Lemma 3.18: Let α be a well-formed sequence of serial operations, and let R be a sibling order. Then if affects$_T(\alpha)$ is consistent with R, affects(α) is consistent with R_E^*.

Proof: We assume without loss of generality that R is a total sibling order. It will be helpful to think of the transitive closure of affects(α) $\cup$ R_E^* as a graph G whose vertices are the events of α and whose edges are the elements of R_E^* and directly-affects(α). Suppose that G contains a cycle. By Lemma 3.13 affects(α) is irreflexive, so some edge (ϕ,π) of the cycle must be in R_E^*. Let T_1 and T_2 be the transactions mentioned by ϕ and π, respectively.

We now divide G into two parts. Let G_1 be the subgraph of G which contains all events τ such that $(\tau,\pi) \in R_E^*$. Let G_2 be the subgraph of G which contains the rest of the events in α. Clearly, $\phi \in G_1$; since R_E^* is irreflexive, $\pi \notin G_1$, so $\pi \in G_2$. By Definition 2.10, an operation τ is in G_1 if and only if there exists a pair of siblings (U,V) in R such that τ mentions a descendant of U and V is an ancestor of T_2. Thus the least common ancestor of two vertices from different subgraphs must mention a proper ancestor of T_2.

Now, if P is a path from π to ϕ in G, it must contain an edge (ψ,σ) such that ψ is a vertex in G_2 and σ is a vertex in G_1. Furthermore, ψ and σ must mention different transactions (as R_E^* does not distinguish between different events mentioning the same transaction). Let (ψ,σ) be the first such edge in P.

Suppose $(\psi,\sigma) \in R_E^*$. As σ is in G_1, (σ,π) must be in R_E^*. Then by transitivity of R_E^*, (ψ,π) is also in R_E^*. But then ψ is in G_1, contrary to choice.

Alternatively, $(\psi,\sigma) \in$ directly-affects(α). Then ψ mentions a proper ancestor U of T_2, and σ mentions a child V of U. Consider the longest subpath P' of P which ends in (ψ,σ) and contains only vertices which mention ancestors of T_2. Then P' is directly preceded by an edge (ρ,τ) where τ mentions an ancestor of T_2 and ρ does not.

There are three cases: either $(\rho,\tau) = (\psi,\sigma)$, $(\rho,\tau) \in R_E^*$, or $(\rho,\tau) \in$

directly-affects(α). In the first case, ψ would not mention an ancestor of T_2, a contradiction. In the second case, ρ would be in G_1, contradicting our choice of (ψ,σ). In the third case, the transaction Z mentioned by ρ affects V in α; by our observation lca(Z,V) is an ancestor of T_2 with children Z',V' which are ancestors of Z and V, respectively. If $(Z',V') \in R$, then $(\rho,\sigma) \in R_E^*$ and ρ is in G_1, a contradiction. On the other hand, if $(V',Z') \in R$, then R is not consistent with affects$_T$(α).

3.3 Serial Correctness

In this section, we describe a general method for constructing serial schedules from schedules of the generic system given a transaction and a sibling order which is consistent with affects$_T$. Starting from a generic schedule α, a sibling order R, and a designated transaction T, our method will have three steps. First, we remove all events from α which cannot affect T, either through the serial controller and transactions (as revealed by affects(α)), or through objects (as revealed by R). Second, we add in ABORT operations to this subsequence to mask transactions which were created in α, but whose returns either had not yet occurred, or were removed in the first step. Finally, we reorder the resulting sequence according to affects(α) $\cup$ $R_E^*|\alpha$. Provided R^* timestamp-orders $\alpha|G(X)$ for each generic object G(X), the final product will be a serial schedule.

3.3.1 Removing Extraneous Events

Let α be a generic schedule and T a transaction. We already have, in affects(α), a tool for detecting what events must be in any well-formed schedule of the serial controller containing $\alpha|T$. Unfortunately affects(α) cannot detect those events which, because they are revealed to T through the actions of a generic object, must also appear in a serial schedule containing T.

It is not difficult to construct an example which illustrates this problem. Suppose that T_1 and T_2 are siblings mentioned in a generic schedule α, and that a COMMIT but no REPORT-COMMIT for T_1 appears in α. Now, because no report for T_1 appears in α, we know by Lemma 3.16 that (T_1,T_2) is not in affects$_T(\alpha)$, and thus that no event of T_1 affects any event of T_2. On the other hand, T_1 has committed, and may be visible to T_2 at some object X in the system. If descendants of both T_1 and T_2 are accesses of G(X), both of which are mentioned in done(α)|G(X), then it is possible that any serial schedule containing the events of T_2 (and, through closure under affects(α), the events of its descendant) would also need to contain the events of the descendant of T_1, and thus some or all of the events of T_1 itself. Whether or not it would depends on whether T_1 precedes T_2 in the sibling order whose descendant closure is used to timestamp-order G(X). In general, this difficulty can arise whenever a transaction has committed, but its COMMIT has not been reported to its parent before a REQUEST-CREATE is issued for one of its siblings. We resolve it by combining affects(α) with a narrowly-defined event order based on the sibling order.

If R is a sibling order, we define $R_{C\text{-}C}$ to be the binary relation on events given by the rule $(\phi,\pi) \in R_{C\text{-}C}$ if and only if

1. ϕ is a COMMIT event for a transaction T_1,
2. π is a CREATE event for a transaction T_2, and
3. (T_1,T_2) is in R.

It is not difficult to see that $R_{C\text{-}C}$ is a subset of R^*_E, so by Lemma 3.18 $R_{C\text{-}C}$ is consistent with affects(α) if R is consistent with affects$_T(\alpha)$. The transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$ retains some of the properties of affects(α), as described below.

Lemma 3.19: Let α be a sequence of serial operations, R a sibling order, and T a transaction. Let ϕ and π be events of α such that ϕ mentions a descendant of T and π does not. Then if (ϕ,π) is in the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$, there is a return event ρ for T such that either $\phi = \rho$, or (ϕ,ρ) and (ρ,π) are both in the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$.

Proof: If (ϕ,π) is in the the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$ there must exist a sequence $\phi,\phi_1,\phi_2,\ldots,\pi$ in which each adjoining pair is in either directly-affects(α) or $R_{C\text{-}C}$. Since ϕ mentions a descendant of T and π does not, there must be some first pair ϕ_n, ϕ_{n+1} where ϕ_n mentions a descendant of T and ϕ_{n+1} does not. If $(\phi_n,\phi_{n+1}$ in directly-affects(α) then ϕ_n must be a return event for T, and ϕ_{n+1} is either a corresponding report event or a COMMIT event for parent(T). Alternatively, if (ϕ_n,ϕ_{n+1}) is in $R_{C\text{-}C}$, then ϕ must be a COMMIT event for T and ϕ_{n+1} must be a CREATE event for some other transaction. In either case, let $\rho = \phi_n$; either $\rho = \phi$, or (ϕ,ρ) and (ρ,π) are both in the the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$. Thus the result holds.

The following lemma differs slightly from Lemma 3.12, in that the addition of $R_{C\text{-}C}$ allows us to bypass REQUEST-CREATE events in paths to CREATE events.

Lemma 3.20: Let α be a sequence of serial operations, R a sibling order, and T a transaction. Let ϕ and π be events of α such that ϕ does not mention a descendant of T and π does. Then if (ϕ,π) is in the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$, there is an instance ρ of CREATE(T) such that either $\pi = \rho$, or (ϕ,ρ) and (ρ,π) are both in the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$.

Proof: If (ϕ,π) is in the the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$ there must exist a sequence $\phi,\phi_1,\phi_2,\ldots,\pi$ in which each adjoining pair is in either directly-affects(α) or $R_{C\text{-}C}$. Since ϕ does not mention descendant of T and π does, there must be some first pair ϕ_n, ϕ_{n+1} where ϕ_n does not mention a descendant of T and ϕ_{n+1} does. If $(\phi_n,\phi_{n+1}$ in directly-affects(α) then ϕ_n must be an instance of REQUEST-CREATE(T), and ϕ_{n+1} must be an instance of CREATE(T). Alternatively, if (ϕ_n,ϕ_{n+1}) is in $R_{C\text{-}C}$, then ϕ must be a COMMIT event for some transaction other than T and ϕ_{n+1} must be an instance of CREATE(T). In either case, let $\rho = \phi_{n+1}$; either $\rho = \pi$, or (ϕ,ρ) and (ρ,π) are both in the the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$. Thus the result holds.

When α is a generic schedule, T a transaction, and R a sibling order consistent with affects(α), we will denote closure(α|T, α, affects(α) $\cup$ $R_{C\text{-}C}$) by core(α,T,R). Certain properties of core(α,T,R) will be important to our proof of serial correctness, and are summarized in the following lemma.

Lemma 3.21: Let α be a generic schedule, R a sibling order consistent with affects$_T$(α), and T a transaction. Then all of the following conditions are true.

1. Let T' be a transaction which is not an ancestor of T. Then if no return event for T' appears in α, no event mentioning a descendant of T' appears in core(α,T,R).

2. Let T' be an arbitrary transaction. Then core(α,T,R)|T' is a prefix of α|T'.

3. core(α,T,R) contains only serial events.

Proof:

1. Let ϕ be an event mentioning a descendant of T'. T is not a descendant of T'; so if no return for T' appears in α, by Lemma 3.19, there cannot be any event π of T such that (ϕ,π) is in the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$. Thus ϕ is not in core(α,T,R).

2. If ϕ and π are events of T' and ϕ precedes π in α, ϕ affects π in α. Thus if any event of T' is in core(α,T,R), all previous events of T' are also in core(α,T,R).

3. Suppose that some non-serial operation ϕ appears in core(α,T,R). Then either ϕ appears in α|T, or there is some event π in α|T such that (ϕ,π) is in the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$. But any event of T is serial, and the domain of the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$ includes only serial events. Thus ϕ cannot appear in core(α,T,R).

3.3.2 Virtual Aborts

As the generic controller allows transactions to return before their children, it is possible that core(α,T,R) contains REQUEST-CREATE events for transactions which do not return in core(α,T,R). The presence of these REQUEST-CREATE events will create difficulties with the serial controller, which requires a return from all transactions whose creation has been requested before their parent may return. We may not remove any REQUEST-CREATEs, because that would disrupt core(α,T,R)|T' for some T'; instead we add ABORT events to core(α,T,R).

Let core^+(α,T,R) be the sequence core(α,T,R)ABORT(T_1)ABORT(T_2)...ABORT(T_n), where T_1 through T_n enumerate the transactions which satisfy the following criteria:

1. T_i is not an ancestor of T.
2. An instance of REQUEST-CREATE(T_i) appears in core(α,T,R).
3. No return event for T_i appears in core(α,T,R).

Lemma 3.22: Let α be a generic schedule, R a sibling order consistent with affects_T(α), and T a transaction. Then all of the following conditions are true.

1. If T' is an ancestor of T, core^+(α,T,R) contains no return event for T'.
2. If T' is not an ancestor of T and core^+(α,T,R) contains any event mentioning T', then core^+(α,T,R) contains exactly one return event for T'.
3. If core^+(α,T,R) contains an instance of ABORT(T'), core^+(α,T,R) contains no other events mentioning T'.
4. For any T', core^+(α,T,R)|T' is a prefix of α|T'.
5. core^+(α,T,R) is well-formed for transactions and basic objects.
6. affects(core^+(α,T,R)) is consistent with R_E^*|α.

Proof:

1. Suppose core$^+(\alpha,T,R)$ contains a return event ϕ for an ancestor T' of T. Then ϕ affects in α some event π of T; but ϕ directly-affects only events which do not mention a descendant of T'. By Lemma 3.12 ϕ affects an instance of CREATE(T'). Then ϕ affects itself in α, and affects(α) is not a partial order. But α is well-formed by Lemma 3.9, and thus affects(α) is a partial order by Lemma 3.13.

2. As core(α,T,R) is a subsequence of α, a schedule of the generic system, core(α,T,R) can contain at most one return event for T'. If core(α,T,R) contains no return event for T', by Lemma 3.21 core(α,T,R) contains no events which mention a descendant of T'. Then either core(α,T,R) does not contain an instance of REQUEST-CREATE(T'), in which case our claim holds; or it does, in which case core$^+(\alpha,T,R)$ will add in an instance of ABORT(T'), and our claim will still hold.

3. Suppose core$^+(\alpha,T,R)$ contains an instance ϕ of ABORT(T'). If ϕ is not in core(α,T,R), then core(α,T,R) contains no return event for T'; thus core(α,T,R) contains no events mentioning a descendant of T' by Lemma 3.21, and the only event of core$^+(\alpha,T,R)$ which mentions T' will then be ϕ. Alternatively, if ϕ is in core(α,T,R), ϕ must be in α. Now, α is a well-formed schedule of the generic system, so by Lemma 3.2 α cannot contain any other return operation for T'; in particular, α cannot contain a COMMIT for T'. Thus by Lemma 3.19 no event of T' other than ϕ can affect any event of T in α, and ϕ is thus the only event mentioning T' in core$^+(\alpha,T,R)$.

4. Core$^+(\alpha,T,R)$ contains all of the events of core(α,T,R); furthermore, all new events in core$^+(\alpha,T,R)$ are not events of any transaction. Thus core$^+(\alpha,T,R)|T'$ = core$(\alpha,T,R)|T'$ for all T', and the claim follows immediately from Lemma 3.21.

5. By the above, the restriction of core$^+(\alpha,T,R)$ to any transaction T' is a prefix of $\alpha|T'$. Since α is well-formed, $\alpha|T'$ is well-formed, and any prefix of $\alpha|T'$ is well-formed. Now, we have made no requirement that T' not be an access transaction; thus $\alpha|X$ is well-formed for X by virtue of the fact that it is well-formed for every access of X, in accordance with Definition 2.4. Thus core$^+(\alpha,T,R)$ is well-formed.

6. Core$^+(\alpha,T,R)$ is a well-formed sequence of serial operations. Thus by Lemma 3.13, affects(core$^+(\alpha,T,R)$) is a partial order. Now, all events in core$^+(\alpha,T,R)$ which are not in α are ABORT events which cannot have any corresponding REPORT-ABORT events, so the only pairs of events (ϕ,π) in directly-affects(core$^+(\alpha,T,R)$) $\cup$ $R_E^*|\alpha$ which are not in directly-affects(α) $\cup$ $R_E^*|\alpha$ will either have ϕ be an instance of ABORT(T'), and π a COMMIT for parent(T'); or ϕ an instance of REQUEST-CREATE(T') and π an instance of ABORT(T'), where T' is a transaction for which an abort was introduced in the formation of core$^+(\alpha,T,R)$ from core(α,T,R). Now, if affects(core$^+(\alpha,T,R)$) is not consistent with $R_E^*|\alpha$, there must be a cycle in

directly-affects(core$^+$(α,T,R)) $\cup$ $R_E^*|\alpha$; as affects(α) is consistent with $R_E^*|\alpha$, this cycle must contain one of the "new" edges in affects(core$^+$(α,T,R)) - affects(α). Suppose a cycle contains the new edge from an instance ϕ of REQUEST-CREATE(T') to an instance π of ABORT(T'). Then the next edge must be from π to a COMMIT event ρ for T'. But α is a generic schedule, so by Lemma 3.2 if α contains a COMMIT event for T' it must also contain a corresponding REQUEST-COMMIT event σ for T'. But by well-formedness of α ϕ must precede (and thus affects) σ, which in turn affects ρ. Clearly σ is in core$^+$(α,T,R); thus we can replace the edges (ϕ,π) and (π,ρ) in our cycle with the edges (ϕ,σ) and (σ,ρ), both of which are in affects(α). In this manner we can convert any cycle in affects(core$^+$(α,T,R)) $\cup$ $R_E^*|\alpha$ into a cycle in affects(α) $\cup$ $R_E^*|\alpha$. The latter is acyclic; thus the former must also be acyclic.

3.3.3 Reordering

By the preceding lemma, affects(core$^+$(α,T,R)) $\cup$ R_E^* is a partial order on core$^+$(α,T,R). Therefore, reorder(core$^+$(α,T,R),affects(core$^+$(α,T,R)) $\cup$ R_E^*) exists; for brevity, we will refer to it as view(α,T,R).

The sequence view(α,T,R) has some useful properties which we state here for reference.

Lemma 3.23: Let α be a generic schedule, T a transaction, and R a sibling order consistent with affects$_T$(α). Then all of the following hold:

1. For any T', view(α,T,R)|T' = core$^+$(α,T,R)|T'.
2. view(α,T,R) is well-formed for transactions and basic objects.

Proof:

1. Because view(α,T,R) is a reordering of core$^+$(α,T,R), it contains the same events as core$^+$(α,T,R). Now, if ϕ and π are events of T' in core$^+$(α,T,R), (ϕ,π) $\in$ affects(core$^+$(α,T,R)) if and only if ϕ precedes π in core$^+$(α,T,R). But view(α,T,R) is affects(core$^+$(α,T,R))-ordered, so then ϕ precedes π in view(α,T,R) if and only if ϕ precedes π in core$^+$(α,T,R). Thus the same events of T' occur in the same order in view(α,T,R) and core$^+$(α,T,R), and thus view(α,T,R)|T' = core$^+$(α,T,R)|T'.
2. By the preceding argument view(α,T,R)|T' = core$^+$(α,T,R)|T', which is well-formed for T' by Lemma 3.22. Furthermore, because our choice of T' did not exclude access transactions, view(α,T,R)|T' is well-formed for any access transaction T'. Thus if X is a basic object, view(α,T,R)|X is also well-formed.

Corollary 3.24: If α is a generic schedule, T a non-access transaction, and R a sibling order consistent with $\text{affects}_T(\alpha)$, then for any non-access transaction T', view(α,T,R)|T' is a schedule of T'.

Proof: By the preceding Lemma, view(α,T,R)|T' is equal to $\text{core}^+(\alpha$,T,R)|T'; but by Lemma 3.22 $\text{core}^+(\alpha$,T,R)|T' is a prefix of α|T', which is a schedule of T'.

We now consider the effect of our procedure the schedules of generic objects. The following definition will be useful in describing the status of accesses in schedules of the system.

Definition 3.25: Let α be an arbitrary sequence of operations, and let T be a transaction and U an ancestor of T. Then we say T is *committed to* U in α if α contains a COMMIT for every ancestor T' of T which is a proper descendant of U.

Lemma 3.26: Let α be a generic schedule, T a transaction which is not an orphan in α, and R a total sibling order consistent with $\text{affects}_T(\alpha)$. Let G(X) be a generic object. Then there exists a sequence β of INFORM-ABORT-AT(X) operations such that (α|G(X))β is well-formed, and, for any access T' of G(X) not equal to T, done((α|G(X))β)|T' is nonempty if and only if T' is committed to lca(T,T') in α.

Proof: Let $U_1,U_2,...,U_n$ enumerate the set of transactions which satisfy the following criteria:

1. U_i is an ancestor of some access U_i of G(X) which is mentioned in α.

2. U_i is not an ancestor of T.

3. No COMMIT for U_i appears in α.

Let β be the sequence INFORM-ABORT-AT(X)OF(U_1)INFORM-ABORT-AT(X)OF(U_2)... INFORM-ABORT-AT(X)OF(U_n). First we must show that (α|G(X))β is well-formed. Assume otherwise. By Lemma 3.9 α|G(X) is well-formed; thus (α|G(X))β will be well-formed unless there is an INFORM-ABORT operation in β for some transaction U_i for which there is an INFORM-COMMIT in α|G(X). But if INFORM-COMMIT-AT(X)OF(U_i) appears in α, then by lemma 3.2 a COMMIT for U_i must appear in α. Thus no INFORM-ABORT for U_i can appear in β, because it does not meet the necessary conditions.

Now suppose T' is an access of G(X) not equal to T which is committed to lca(T,T') in α. Then α contains a COMMIT for T', and thus by Lemma 3.2 α contains a REQUEST-COMMIT for T'. Thus done((α|G(X))β)|T' will be nonempty provided (α|G(X))β contains no INFORM-ABORT operations for any ancestor U of T'. Suppose that there is actually some such U. There are then two cases. If U is a proper descendant of lca(T,T'), then INFORM-ABORT-AT(X)OF(U) cannot appear in β, because a COMMIT for U appears in α. Thus INFORM-ABORT-AT(X)OF(U) must appear in α. But then by Lemma 3.2 α must contain an ABORT for U, which contradicts the result of Lemma 3.2 that α

contains at most one return for any transaction. Thus U cannot be a proper descendant of lca(T,T').

The alternative is that U is an ancestor of lca(T,T'), and thus of T. Because U is an ancestor of T, no INFORM-ABORT for U can appear in β; but if an INFORM-ABORT for U appears in α, then an ABORT for U must also appear in α, which would make T an orphan, contrary to hypothesis. Thus $(\alpha|G(X))\beta$ does not contain an INFORM-ABORT for any ancestor U of T', and done($(\alpha|G(X))\beta$) is nonempty.

Conversely, suppose there exists some ancestor U of T' which is a proper descendant of lca(T,T') and for which no COMMIT appears in α. Then either α|T' is empty, in which case done($(\alpha|G(X))\beta$)|T' is also empty; or U is an ancestor of an access of G(X) mentioned in α. But U is not an ancestor of T, and no COMMIT for U appears in α. Thus an INFORM-ABORT-AT(X)OF(U) appears in β, and done($(\alpha|G(X))\beta$)|T' is the empty sequence.

Lemma 3.27: Let α be a generic schedule, T a transaction which is not an orphan in α, and R a total sibling order consistent with affects$_T(\alpha)$. Let G(X) be a generic object, and let β be as in Lemma 3.26. Then if ϕ is an event of rearrange($(\alpha|G(X))\beta,R^*$) which appears in view(α,T,R), every event which precedes ϕ in rearrange($(\alpha|G(X))\beta,R^*$) also appears in view(α,T,R).

Proof: Let T_1 = transaction(ϕ). Let π be an event which precedes ϕ in rearrange($(\alpha|G(X))\beta,R^*$). Then either π and ϕ are events of the same transaction, in which case π affects ϕ in α and thus π appears in view(α,T,R), or π is an event of an access T_2 such that (T_2,T_1) is in R^*. Note that, by the conditions of Lemma 3.26, T_1 is committed to lca(T_1,T) and T_2 is committed to lca(T_2,T) in α. Now, either lca(T_2,T_1) is an ancestor of lca(T_1,T) or vice versa. In the first case, lca(T_2,T_1) = lca(T_2,T) and T_2 is committed to lca(T_2,T_1) in α. In the second case, lca(T_1,T) = lca(T_2,T); T_2 is committed to lca(T_2,T) in α, so it must be committed to lca(T_2,T_1).

Let U_2 be the ancestor of T_2 which is a child of lca(T_2,T_1), and U_1 the corresponding ancestor of T_1. Since T_2 is committed to lca(T_2,T_1), there is a chain of COMMIT events in α for ancestors of T_2 up to and including U_2, each of which is affected by the REQUEST-COMMIT event ρ for T_2 in rearrange($(\alpha|G(X))\beta,R^*$). Now, by repeated application of Lemma 3.2 and well-formedness we can show that there is a similar chain of REQUEST-CREATE and CREATE events for ancestors of T_1 up to and including U_1, each of which affects the CREATE event σ for T_1 in rearrange($(\alpha|G(X))\beta,R^*$). Now, since (T_2,T_1) is in R^*, (U_2,U_1) must be in R. Thus there is an edge in R_{C-C} from the COMMIT for U_2 to the CREATE for U_1, and thus (ρ,σ) is in the the transitive closure of affects(α) $\cup$ R_{C-C}. Now, π can either be σ or a REQUEST-COMMIT for T_1, and ϕ can either be ρ or a CREATE for T_2; in each case (ϕ,π) is in the the transitive closure of affects(α) $\cup$ R_{C-C}. Thus ϕ appears in view(α,T,R) if π does.

Lemma 3.28: Let α be a generic schedule, T a transaction which is not an orphan in α, R a total sibling order which is consistent with affects$_T(\alpha)$, and G(X) a generic object. Then view(α,T,R)|X is a prefix of rearrange($(\alpha|G(X))\beta,R^*$), where β is as in Lemma 3.26.

Proof: First we show that view(α,T,R)|X cannot contain any event ϕ which does not appear in rearrange((α|G(X))β,R*). Suppose otherwise. Then ϕ is an event of some access T' of G(X) which has an ancestor U which is a proper descendant of lca(T,T') and which is not committed in α. If U is aborted in α, then ϕ cannot appear in view(α,T,R) by Lemma 3.22. Alternatively, if no return event for U appears in α, then by Lemma 3.19 there can be no event π of T such that (ϕ,π) is in the the transitive closure of affects(α) $\cup$ $R_{C\text{-}C}$. In either case, ϕ cannot appear in view(α,T,R).

Now, by Lemma 3.27 if any event in rearrange((α|G(X))β,R*) appears in view(α,T,R), all preceding events in rearrange((α|G(X))β,R*) appear in view(α,T,R). Thus the set of events in view(α,T,R)|X is equal to the set of events of some prefix of rearrange((α|G(X))β,R*). But both view(α,T,R)|X and rearrange((α|G(X))β,R*) are ordered by R_E^*; thus view(α,T,R)|X) is, in fact, equal to rearrange((α|G(X))β,R*).

Corollary 3.29: Let α, T, R, G(X) and β be as above. Then if R* timestamp-orders α|G(X), view(α,T,R)|X is a schedule of X.

Proof: If R* timestamp-orders α|G(X), then, by Definition 3.7 and Lemma 3.26, rearrange((α|G(X))β,R*) is a schedule of X. By Lemma 3.28 view(α,T,R)|X is a prefix of rearrange((α|G(X))β,R*). Thus view(α,T,R)|X is also a schedule of X.

Finally, we show that view yields a schedule of the serial controller.

Lemma 3.30: Let α be a generic schedule, T a transaction, and R a total sibling order consistent with affects$_T$(α). Then view(α,T,R) is a well-formed schedule of the serial controller.

Proof: Well-formedness is guaranteed by Lemma 3.23. We now proceed by showing by induction that any prefix of view(α,T,R) is a schedule of the serial controller.

Let β be a prefix of view(α,T,R). If β is the empty sequence, β is trivially a schedule of the serial controller. Otherwise, let β = β'π where π is a single operation. By induction hypothesis β' is a schedule of the serial controller. Let s' be a state of the serial controller which can follow β'. We demonstrate for each possible value of π that all of the preconditions for a transition whose operation is π are met in s', and that β is thus a schedule of the serial controller.

- If π is a REQUEST-CREATE or REQUEST-COMMIT event, then no preconditions need to be met, and β is a schedule of the serial controller.

- If π is an instance of CREATE(T'), then there must exist an instance ϕ of REQUEST-CREATE(T') in α; ϕ affects π in α, so ϕ appears in core$^+$(α,T,R). Since ϕ also affects π in core$^+$(α,T,R), ϕ must precede π in view(α,T,R). Thus ϕ appears in β', and T' $\in$ s'.create-requested. For the second precondition, suppose that there exists a sibling U of T' such that an instance of CREATE(U) appears in β' but no return event for U appears in β'. By Lemma 3.22, view(α,T,R) must contain exactly one return operation for every transaction mentioned in view(α,T,R) which is not an ancestor of T. There are three possible cases:

1. Neither T' nor U are ancestors of T. Then by Lemma 3.22 $\mathrm{core}^{+}(\alpha,T,R)$ must contain COMMIT events for both T' and U. Now, R is a total sibling order, so either $(T',U) \in R$ or $(U,T') \in R$. If $(T',U) \in R$, then $R_{C\text{-}C}$ contains an edge from any COMMIT for T' to any CREATE for U, and thus the COMMIT event for T' must precede the instance of CREATE(U) in β'. But then the COMMIT event would precede π, and view(α,T,R) would not be ordered by affects$(\mathrm{core}^{+}(\alpha,T,R))$.

2. T' is an ancestor of T. Then a COMMIT event ϕ for U appears in view(α,T,R). By Lemma 3.20, if there is any event ρ of T such that (ϕ,ρ) is in the transitive closure of affects$(\alpha) \cup R_{C\text{-}C}$, then there is some instance σ of CREATE(T) in $\mathrm{core}^{+}(\alpha,T,R)$ such that (ϕ,σ) is in the transitive closure of affects$(\alpha) \cup R_{C\text{-}C}$. But view$(\alpha,T,R)$ is well-formed, so at most one CREATE(T) event appears in view(α,T,R); thus $\sigma = \pi$, and ϕ must be in β'.

3. U is an ancestor of T. Let ϕ be the instance of CREATE(U) in β'. π is in view(α,T,R); so there is some event ρ of T such that (π,ρ) is in the transitive closure of affects$(\alpha) \cup R_{C\text{-}C}$. But then by the argument used in the previous case, (π,ϕ) must be in the transitive closure of affects$(\alpha) \cup R_{C\text{-}C}$. But then π would precede ϕ in view(α,T,R), a contradiction.

Thus both preconditions are met, and β is a schedule of the serial controller.

- If π is an instance of ABORT(T'), then either π is in core(α,T,R) or π was added in forming $\mathrm{core}^{+}(\alpha,T,R)$. In the first case, α contains π, so α must contain an instance ϕ of REQUEST-CREATE(T'). ϕ affects π in α, so ϕ is in core(α,T,R) and thus in view(α,T,R). Furthermore, ϕ affects π in $\mathrm{core}^{+}(\alpha,T,R)$; so ϕ precedes π in view(α,T,R), and ϕ thus appears in β'. In the second case, an instance ϕ of REQUEST-CREATE(T') must appear in core(α,T,R); otherwise π would not have been added. Again ϕ must precede π in view(α,T,R). Thus in either case $T' \in s'$.create-requested. Now, by Lemma 3.22, $\mathrm{core}^{+}(\alpha,T,R)$ can contain no other event mentioning T'; thus view(α,T,R) contains no CREATE(T') events, and T' cannot be in s'.created. So the precondition is met.

- If π is an instance of REPORT-ABORT(T'), then π must appear in α. Thus by Lemma 3.2, an instance ϕ of ABORT(T') appears in α. As ϕ affects π in α, ϕ must precede π in view(α,T,R). Thus $T' \in s'$.aborted, the sole precondition on ABORT(T') is met, and β is a schedule of the serial controller.

- If π is an instance of COMMIT(T',v), then π must appear in α. By Lemma 3.22, T' cannot be an ancestor of T. By Lemma 3.2 an instance ϕ of REQUEST-COMMIT(T',v) also appears in α. As ϕ affects π in α, ϕ must precede π in view(α,T,R); thus $(T',v) \in s'$.commit-requested. By Lemma 3.22, exactly one return event for T' appears in $\mathrm{core}^{+}(\alpha,T,R)$; thus no return event for T' can occur in β', and T' $\notin$ s'.returned. Finally, suppose there is some child U of T' such that an instance of REQUEST-CREATE(U) appears in β', but no return event for U appears in β'. Any return event for U affects π in α, so if no return event for U appears in β', none appears in α. Thus there is a REQUEST-CREATE event but no return event for U in core(α,T,R). Also, U is not an ancestor of T, so core$^+$(α,T,R) will contain an ABORT event ρ for U. But ρ affects π in core$^+$(α,T,R), so ρ must appear in β'. Thus children(T') $\cap$ s'.create-requested $\subseteq$ s'.returned, as required by the third and final precondition on COMMIT(T',v).

- If π is an instance of REPORT-COMMIT(T',v), then π must appear in α. Then α must contain both an instance ϕ of COMMIT(T',v) and an instance ρ of REQUEST-COMMIT(T',v). As both ϕ and ρ affect π in α, both ϕ and ρ must precede π in view(α,T,R). Thus (T',v) $\in$ s'.commit-requested and T' $\in$ s'.committed, and, all preconditions on REPORT-COMMIT(T',v) being met, β is a schedule of the serial controller.

We can now combine our results into a single theorem.

Theorem 3.31: Let α be a generic schedule. If there exists a total sibling order R which is consistent with affects$_T$(α), such that R^* timestamp-orders $\alpha|G(X)$ for every generic object G(X), then α is serially correct for all non-orphan transactions.

Proof: Let T be an arbitrary non-orphan transaction. Then core(α,T,R)|T = α|T, so by Lemma 3.23 view(α,T,R)|T = α|T. Now, if T' is any other transaction, by Corollary 3.24 view(α,T,R)|T' is a schedule of T'. If X is a basic object, then by Corollary 3.29 view(α,T,R)|X is a schedule of X. Finally, Lemma 3.30 guarantees that view(α,T,R) is a schedule of the serial controller. So, for any component *A* of the serial system, view(α,T,R)|*A* is a schedule of *A*; and thus view(α,T,R) is a serial schedule.

Because our choice of T was arbitrary, for any non-orphan transaction T, view(α,T,R)|T = α|T and view(α,T,R) is a serial schedule. Consequently, α is serially correct for any non-orphan transaction T.

Chapter 4

The Pseudotime System

In this chapter we describe an implementation of a system whose schedules meet all of the conditions of Theorem 3.31. The system is based on the pseudotime algorithm described in [R].

The essential feature of Reed's algorithm is the use of a totally-ordered set of *pseudotimes* to regulate interaction between concurrently executing transactions. Before a transaction is created, the system assigns to it a contiguous range of pseudotime which is a subset of the range of its parent, and which is disjoint from any ranges already assigned to its siblings. Disjointness allows us to derive a sibling order from the pseudotime ordering in a natural way; the requirement that a transaction receive a subrange of its parent's range ensures that comparisons of disjoint pseudotime ranges yields an ordering on transactions which corresponds to the descendant closure of this sibling order.

Formally, we let P be the set of pseudotimes, ordered by <. We represent pseudotime ranges as half-open intervals $[p,q)$ in P, and refer to them using capital letters. If $\mathrm{P} = [p,q)$, then we write $\mathrm{P}_{\mathrm{min}}$ for p and $\mathrm{P}_{\mathrm{max}}$ for q. If P and Q are ranges of pseudotime, we write $\mathrm{P} < \mathrm{Q}$ if $\mathrm{P}_{\mathrm{max}} \leq \mathrm{Q}_{\mathrm{min}}$. Clearly, if P < Q, then P and Q are disjoint.

It will be convenient to extend P with two dummy pseudotimes, $-\infty$ and $+\infty$. We write $P^+ = P \cup \{-\infty,+\infty\}$; if $p, q \in P^+$, we let $p < q$ if and only if $p < q$ in P, $p = -\infty$, or $q = +\infty$. Thus $-\infty$ is a pseudotime less than all others, and $+\infty$ is a pseudotime greater than all others. Unless otherwise stated, when we refer to a pseudotime we will be referring to an element of P^+ rather than P.

The pseudotime system will consist of a pseudotime controller, pseudotime objects, and transactions. The transactions will be the same as in the serial and generic systems; the pseudotime controller and objects will differ in their external behavior from the corresponding components of the generic system only by the addition of a new, ASSIGN-PSEUDOTIME operation. Thus if α is a schedule of the pseudotime system, generic(α) will be a schedule of the generic system, allowing us to use Theorem 3.31 to prove serial correctness.

4.1 Pseudotime Controller

A state s of the pseudotime controller, like one of the generic and serial controllers, has components s.created-requested, s.created, s.commit-requested, s.committed, and s.aborted; in addition, a state of the pseudotime controller has a component s.range, which is a partial function from *T* to the set of pseudotime ranges. In the initial state of the controller, create-requested = $\{T_0\}$, range = $\{<T_0,P_0>\}$ for some pseudotime range P_0, and all other components are empty. The operations of the pseudotime controller are as follows:

Input:
REQUEST-CREATE(T), $T \neq T_0$
REQUEST-COMMIT(T,v)
Output:
ASSIGN-PSEUDOTIME(T,P), P a pseudotime range, $T \neq T_0$
CREATE(T)
ABORT(T), $T \neq T_0$
COMMIT(T,v), $T \neq T_0$
REPORT-ABORT(T), $T \neq T_0$
REPORT-COMMIT(T,v), $T \neq T_0$
INFORM-ABORT-AT(X)OF(T), $T \neq T_0$
INFORM-COMMIT-AT(X)OF(T), $T \neq T_0$

The steps of the pseudotime controller are those tuples (s',π,s) which satisfy the following pre- and postconditions.

- REQUEST-CREATE(T)
  Postcondition:
  s.create-requested = s'.create-requested $\cup$ $\{T\}$
- REQUEST-COMMIT(T,v)
  Postcondition:
  s.commit-requested = s'.commit-requested $\cup$ $\{(T,v)\}$

- ASSIGN-PSEUDOTIME(T,P)
Preconditions:
T $\in$ s'.create-requested
T $\notin$ domain(s'.range)
P $\subseteq$ s'.range(parent(T))
P > s'.range(T') for every T' in siblings(T) $\cap$ domain(s'.range)
Postconditions:
s.range = s'.range $\cup$ {<T,P>}

- CREATE(T)
Preconditions:
T $\in$ s'.create-requested - s'.created
T $\in$ domain(s'.range)
Postcondition:
s.created = s'.created $\cup$ {T}

- ABORT(T)
Precondition:
T $\in$ s'.create-requested - s'.returned
Postconditions:
s.aborted = s'.aborted $\cup$ {T}

- REPORT-ABORT(T)
Precondition:
T $\in$ s'.aborted

- COMMIT(T,v)
Preconditions:
(T,v) $\in$ s'.commit-requested
T $\notin$ s'.returned
Postcondition:
s.committed = s'.committed $\cup$ {T}

- REPORT-COMMIT(T,v)
Preconditions:
(T,v) $\in$ s'.committed-requested
T $\in$ s'.committed

- INFORM-ABORT-AT(X)OF(T)
Precondition:
T $\in$ s'.aborted

- INFORM-COMMIT-AT(X)OF(T)
Precondition:
T $\in$ s'.committed

A careful comparison between the pre- and postconditions of the pseudotime and generic controllers will reveal that the only differences are the addition of the ASSIGN-PSEUDOTIME operation and the addition of a precondition on CREATE(T) which requires that T already have been assigned a pseudotime range. In fact, we can easily prove that schedules of the pseudotime schedule are reducible to schedules of the generic schedule by removing all ASSIGN-PSEUDOTIME operations.

Lemma 4.1: Let α be a schedule of the pseudotime controller which leads to a state s from the initial state. Then generic(α) is a schedule of the generic controller which leads to a state t from the initial state such that:

s.create-requested = t.create-requested
s.created = t.created
s.commit-requested = t.commit-requested
s.committed = t.committed
s.aborted = t.aborted

Proof: If α is the empty sequence, the result holds by virtue of the fact that all components of the initial state of the generic controller are equal to the corresponding components of the initial state of the pseudotime controller. Otherwise, let $\alpha = \alpha'\pi$, where π is a single operation, and suppose that the result holds for α'. Let s' be the state of the pseudotime controller after α', and s the state after α. Let t' be the state of the generic controller after generic(α'). Then if π is a REQUEST-CREATE, REQUEST-COMMIT, ABORT, COMMIT, REPORT-ABORT, REPORT-COMMIT, INFORM-ABORT, or INFORM-COMMIT operation, the result holds for α because the pre- and postconditions on π are the same for both controllers.

Alternatively, π is either ASSIGN-PSEUDOTIME(T,P) for some T,P or CREATE(T) for some T. In the former case, generic(α) = generic(α'), and s differs from s' only in its range component; thus the result holds. In the latter case, the preconditions on π in the pseudotime controller are a superset of the preconditions on π in the generic controller; thus if π is enabled in t' for the pseudotime controller, it must also be enabled in s' for the generic controller, and generic(α) is a schedule of the generic controller. That the state of the pseudotime controller following α matches the state of the generic controller following generic(α) follows from the fact that the postconditions on CREATE(T) in both controllers are the same. Thus in all cases the result holds.

The preceding lemma allows us to infer properties of the pseudotime controller from properties of the generic controller. For example, Lemma 3.1 describes all but the range component of a pseudotime controller state following a particular schedule; and all of the conditions of Lemma 3.2 hold true for pseudotime schedules.

Unfortunately, Lemma 4.1 does not tell us anything about the range component of the controller state, or about ASSIGN-PSEUDOTIME operations which might appear in schedules of the generic controller. As can be seen by a quick examination of the postcondition on the ASSIGN-PSEUDOTIME operation, there is a straightforward correspondence between ASSIGN-PSEUDOTIME operations appearing in a schedule of the pseudotime controller and the transaction-range pairs appearing in the state of the controller following that schedule. It is more convenient for us to work with schedules than states; thus we define the following.

Definition 4.2: Let α be a sequence of operations. Then we define the relation assign_α by the rule $<T,P> \in \text{assign}_\alpha$ if and only if ASSIGN-PSEUDOTIME(T,P) appears in α.

If α is a schedule of the pseudotime controller, assign_α allows us to describe the range component of any state which can follow α without requiring us to actually consider the entire state.

Lemma 4.3: Let α be a schedule of the pseudotime controller which can lead to a state s from the initial state. Then $\text{s.range} = \text{assign}_\alpha \cup \{<T_0,P_0>\}$.

Proof: By induction on α using the controller postconditions.

Lemma 4.3 allows us to use s.range and assign_α interchangeably when α is a schedule which leads to s from the initial state, a fact which we will use extensively when we prove properties of assign_α for schedules of the pseudotime controller. The benefits of using assign_α instead of s.range will become apparent in our discussion of pseudotime objects, when we consider sequences of operations which might contain ASSIGN-PSEUDOTIME operations, but which will not, in general, be schedules of the pseudotime controller.

The following lemmas describe some of the properties of assign_α when α is a schedule of the pseudotime controller.

Lemma 4.4: Let α be a schedule of the pseudotime controller. Then all of the following conditions hold.

1. If T_1 and T_2 are siblings in the domain of assign_α then $\text{assign}_\alpha(T_1) < \text{assign}_\alpha(T_2)$ if and only if ASSIGN-PSEUDOTIME(T_1) precedes ASSIGN-PSEUDOTIME(T_2) in α.

2. If T and parent(T) are both in the domain of assign_α, then $\text{assign}_\alpha(T) \subseteq \text{assign}_\alpha(\text{parent}(T))$.

Proof: By induction on α using Lemma 4.3 and the pseudotime controller pre- and postconditions on ASSIGN-PSEUDOTIME.

We have not yet described the relationship between ASSIGN-PSEUDOTIME and other operations in a schedule of the pseudotime controller. We do so in the following lemma.

Lemma 4.5: Let α be a schedule of the pseudotime controller. Then all of the following conditions hold.

1. If CREATE(T) appears in α, then an ASSIGN-PSEUDOTIME for T precedes it in α.
2. If ASSIGN-PSEUDOTIME(T,P) appears in α, then REQUEST-CREATE(T) precedes it in α.
3. If ASSIGN-PSEUDOTIME(T,P) appears in α, and parent(T) $\neq T_0$, then an ASSIGN-PSEUDOTIME for parent(T) precedes it in α.
4. At most one ASSIGN-PSEUDOTIME operation appears in α for each transaction.

Proof: By induction on α using the controller preconditions.

If α is a schedule of the pseudotime controller, we can use assign_α to define a sibling order on the transactions in $\text{domain}(\text{assign}_\alpha)$ which will be consistent with $\text{affects}_T(\alpha)$.

Definition 4.6: Let α be an sequence of operations such that assign_α is a partial function. Then let P_α be the binary relation on transactions defined by the rule $<T_1,T_2> \in P_\alpha$ if and only if T_1 and T_2 are siblings in the domain of assign_α, and $\text{assign}_\alpha(T_1) < \text{assign}_\alpha(T_2)$.

Lemma 4.7: Let α be a schedule of the pseudotime controller. Then P_α is a sibling order consistent with $\text{affects}_T(\alpha)$.

Proof: By Lemma 4.4, if T_1 and T_2 are any two siblings in the domain of assign_α, $\text{assign}_\alpha(T_1)$ and $\text{assign}_\alpha(T_2)$ must be disjoint. Thus P_α inherits transitivity, irreflexivity, and antisymmetry from *P*, and is a sibling order.

Now, suppose that P_α is not consistent with $\text{affects}_T(\alpha)$. Then there must exist some cycle of sibling transactions $T_1,T_2,...,T_n$ where each consecutive pair is either in $\text{affects}_T(\alpha)$ or P_α. Because both P_α and $\text{affects}_T(\alpha)$ are transitive, we may assume without loss of generality that no two consecutive edges are both in $\text{affects}_T(\alpha)$ or P_α. Now, suppose $<T_i,T_{i+1}>$ is in $\text{affects}_T(\alpha)$. Then $<T_{i-1},T_i>$ and $<T_{i+1},T_{i+2}>$ are both in P_α, from which we know that T_i and T_{i+1} are both in $\text{domain}(\text{assign}_\alpha)$. By Lemma 3.16, if $<T_i,T_{i+1}>$ is in $\text{affects}_T(\alpha)$, a report for T_i must precede REQUEST-CREATE(T_{i+1}) in α. Then CREATE(T_i) precedes REQUEST-CREATE(T_{i+1}). Now, we know that ASSIGN-PSEUDOTIME events for both T_i and T_{i+1} appear in α; by Lemma 4.5 the ASSIGN-PSEUDOTIME for T_i must precede CREATE(T_i), and the ASSIGN-PSEUDOTIME for T_{i+1} must follow the REQUEST-CREATE(T_{i+1}). But then an ASSIGN-PSEUDOTIME for T_i precedes an ASSIGN-PSEUDOTIME for T_{i+1}, so by Lemma 4.4 $\text{assign}_\alpha(T_i) < \text{assign}_\alpha(T_{i+1})$, and thus $<T_i,T_{i+1}> \in P_\alpha$. By this means we can replace all edges in $\text{affects}_T(\alpha)$ in our cycle with edges in P_α, yielding a cycle in P_α. But this would contradict our proof that P_α is a partial order. Thus P_α is consistent with $\text{affects}_T(\alpha)$.

Unfortunately, there will be conditions (such as when we consider schedules of pseudotime objects), when we will not have access to a complete schedule of the pseudotime controller. Under these circumstances, the following ordering will be more useful.

Definition 4.8: Let α be an arbitrary sequence of operations. Then P'_α is the relation defined by the rule $<T_1,T_2> \in P'_\alpha$ if and only if T_1 and T_2 are both in the domain of assign_α, and $\text{assign}_\alpha(T_1) < \text{assign}_\alpha(T_2)$.

When α is not a complete schedule of the pseudotime controller, P'_α will be the only ordering on transactions which is readily available. When α is a complete schedule, however, we would much rather use the sibling order P_α. Fortunately, we do not have to choose between them, as P'_α will be included in the descendant closure of P_α.

Lemma 4.9: Let α be a schedule of the pseudotime controller. Then P'_α is a subset of P^*_α.

Proof: Suppose $<T_1,T_2> \in P'_\alpha$. Suppose T_1 is an ancestor of T_2; then by Lemma 4.5 every ancestor of T_2 which is a descendant of T_1 is in domain(assign_α). Then by repeated application of Lemma 4.4, $\text{assign}_\alpha(T_2) \subseteq \text{assign}_\alpha(T_1)$. But then $\text{assign}_\alpha(T_1)$ and $\text{assign}_\alpha(T_2)$ are incomparable, and $<T_1,T_2>$ cannot be in P'_α.

Alternatively, $\text{lca}(T_1,T_2)$ is distinct from both T_1, and T_2. Let U_1 and U_2 be children of $\text{lca}(T_1,T_2)$ which are ancestors of T_1 and T_2 respectively. By Lemmas 4.5 and 4.4 $\text{assign}_\alpha(T_1) \subseteq \text{assign}_\alpha(U_1)$ and $\text{assign}_\alpha(T_2) \subseteq \text{assign}_\alpha(U_2)$; since U_1 and U_2 are siblings, Lemma 4.4 guarantees that $\text{assign}_\alpha(U_1)$ and $\text{assign}_\alpha(U_2)$ are disjoint. But $\text{assign}_\alpha(T_1) < \text{assign}_\alpha(T_2)$, so $\text{assign}_\alpha(U_1) < \text{assign}_\alpha(U_2)$. Thus $<U_1,U_2>$ is in P_α, so $<T_1,T_2>$ is in P^*_α.

When α is a schedule of the pseudotime controller, we can always extend P_α into a total sibling order consistent with $\text{affects}_T(\alpha)$. To meet the conditions of Theorem 3.31, we will also need to know that the descendant closure of this sibling order timestamp-orders generic(α)|G(X) for each object X. We can guarantee this condition is met by a careful implementation of pseudotime objects.

4.2 Pseudotime Objects

The pseudotime system represents each basic object X with a pseudotime object P(X). P(X) has the following operations:

Input:
CREATE(T), T $\in$ accesses(X)
ASSIGN-PSEUDOTIME(T,P), T $\in$ accesses(X)
INFORM-ABORT-AT(X)OF(T)
INFORM-COMMIT-AT(X)OF(T)
Output:
REQUEST-COMMIT(T,v), T $\in$ accesses(X)

4.2.1 Well-formedness

Pseudotime objects possess a well-formedness condition which is an extension of the condition for generic objects.

Definition 4.10: Let α be a sequence of operations of P(X). Then α is well-formed if α is the empty sequence, or if $\alpha = \alpha'\pi$, where α' is well-formed and π is a single operation, and the following conditions are met:

1. If π is CREATE(T), for T in accesses(X), then
   a. CREATE(T) does not appear in α', and

   b. ASSIGN-PSEUDOTIME(T,P) appears in α' for some pseudotime range P.
2. If π is REQUEST-COMMIT(T,v), then
   a. CREATE(T) appears in α', and

   b. REQUEST-COMMIT(T,v') does not appear in α' for any value of v'.
3. If π is INFORM-ABORT-AT(X)OF(T), then
   a. INFORM-COMMIT-AT(X)OF(T) does not appear in α'.
4. If π is INFORM-COMMIT-AT(X)OF(T), then
   a. INFORM-ABORT-AT(X)OF(T) does not appear in α', and

   b. if T $\in$ accesses(X), REQUEST-COMMIT(T,v) appears in α' for some v.
5. If π is ASSIGN-PSEUDOTIME(T,P), then
   a. ASSIGN-PSEUDOTIME(T,P') does not appear in α' for any pseudotime range P'.

   b. For any operation ASSIGN-PSEUDOTIME(T',P') appearing in α', P' is disjoint from P.

The additional conditions are straightforward. Essentially, we require only that each access be assigned a unique pseudotime range, disjoint from those of all other accesses, before it is created. The other conditions, carried over from Definition 3.3, exist to ensure that a well-formed sequence of pseudotime object operations may be transformed in to a well-formed sequence of generic object operations by removing all ASSIGN-PSEUDOTIME operations. We state this fact, for future reference, as the following lemma.

Lemma 4.11: Let α be a well-formed sequence of operations of a pseudotime object P(X). Then generic(α) is a well-formed sequence of operations of the generic object G(X).

Proof: By induction on α.

Note that Lemma 4.11, together with Lemma 3.4, guarantees that if α is a well-formed sequence of operations of P(X), then serial(α) is a well-formed sequence of operations of X. Pseudotime objects are required to preserve well-formedness.

4.2.2 Correctness Condition

When α is a schedule of the pseudotime controller, P_α is the obvious order to use when applying Theorem 3.31; however, because each pseudotime object P(X) has ASSIGN-PSEUDOTIME operations only for accesses to X, P_α is not available in its entirety to the pseudotime objects, and any correctness condition we define must use only the information available in $\alpha|P(X)$. Fortunately, Lemma 4.9 lets us base our correctness condition on P'_α.

Let α be a schedule of P(X). If a transaction T is mentioned in done(α), a REQUEST-COMMIT for T appears in α; thus by well-formedness an ASSIGN-PSEUDOTIME for T appears in α and T is in assign_α. Well-formedness also guarantees that $\text{assign}_\alpha(T_1)$ and $\text{assign}_\alpha(T_2)$ are disjoint for any T_1, T_2 mentioned in done(α). Thus P'_α totally orders the transactions mentioned in α, and rearrange(generic(α),P'_α) exists. Our correctness condition for P(X) can thus be stated as the simple requirement that, for all schedules α of P(X), rearrange(generic(α),P'_α) is a schedule of X.

4.2.3 Reads and Writes

Our implementation of pseudotime objects is closely modeled after the *object history* mechanism described in [R]. In [R], the data objects represented by object histories are simple memory locations, supporting two atomic operations, READ and WRITE. Though the basic objects in the serial system may be much more sophisticated, the essential features of our algorithm will still depend on the identification of accesses with READ-like or WRITE-like properties.

Let α and β be well-formed sequences of operations of a basic object X. Then we say α is *equieffective* to β if, for every sequence γ of operations of X such that both $\alpha\gamma$ and $\beta\gamma$ are well-formed, $\alpha\gamma$ is a schedule of X if and only if $\beta\gamma$ is a schedule of X.

Clearly, α is equieffective to β if and only if β is equieffective to α; in this situation we say α and β are equieffective sequences. If neither α nor β is a schedule of X, they are trivially equieffective. On the other hand, if α is equieffective to β, and β is a schedule of X, then α must also be a schedule of X.

Well-formed extensions of equieffective sequences are equieffective, as shown in the following lemma.

Lemma 4.12: Let α and β be equieffective sequences of operations of the basic object X. Then if γ is a sequence of operations of X such that $\alpha\gamma$ and $\beta\gamma$ are both well-formed, $\alpha\gamma$ is equieffective to $\beta\gamma$.

Proof: Suppose otherwise. Then there exists a sequence of operations δ such that $\alpha\gamma\delta$ and $\beta\gamma\delta$ are both well-formed, but only one of $\alpha\gamma\delta$ and $\beta\gamma\delta$ is a schedule of X. But then $\gamma\delta$ distinguishes α and β, which contradicts their equieffectiveness.

We may use equieffectiveness to define the essential properties of a read access in a natural way.

Definition 4.13: Let T be an access to X. Then T is a *read access* if, for any sequence of operations α and value v such αCREATE(T)REQUEST-COMMIT(T,v) is a well-formed schedule of X, αCREATE(T)REQUEST-COMMIT(T,v) is equieffective to α.

Less formally, a read access is one which cannot be detected by later accesses to X. We now define a write access, one which obscures all preceding accesses.

Definition 4.14: Let T be an access to X. Then T is a *write access* if, for any schedule α of X and value v such that αCREATE(T)REQUEST-COMMIT(T,v) is well-formed, αCREATE(T)REQUEST-COMMIT(T,v) is equieffective to CREATE(T)REQUEST-COMMIT(T,v).

This definition requires a bit of unraveling. Often, CREATE(T)REQUEST-COMMIT(T,v) will *not* be a schedule of X; in this case αCREATE(T)REQUEST-COMMIT(T,v) will also not be a schedule of X. The more interesting case is that in which CREATE(T)REQUEST-CREATE(T,v) is a schedule of X; then αCREATE(T)REQUEST-COMMIT(T,v) is a schedule of X whenever it is well-formed. Thus whether a write access is possible is independent of any preceding operations, so long as well-formedness is preserved. Furthermore, a write determines the future behavior of X, again independently of any preceding operations.

Many accesses will be neither reads nor writes. We refer to accesses in this class as *updates*. Updates are interesting primarily for the properties they do not have. Because they are not reads, they change the state of the object in ways that are detectable by later accesses; because they are not writes, their effects depend on the results of earlier accesses. It will be useful to be able to refer to accesses by these inverse properties. Thus we use the term *writer* to refer to any access that is either a write or an update, and *reader* to refer to any access that is either a read or an update. Using this terminology, updates are both readers and writers, while other accesses are either readers or writers, but not both.

4.2.4 Object Implementation

We construct a pseudotime object P(X) for each object X. Each state s of P(X) has components s.created, s.commit-requested, s.committed, and s.aborted, which are sets of transactions; a component s.versions, consisting of accesses of X; a component s.start which is a mapping from the accesses of X to *P*; a component s.state which is a mapping from accesses of X to states of X; and a component s.end which is a mapping from states of X to *P*.

We will use a dummy transaction T_X, assumed to be always visible to all transactions, to represent the version associated with the start state of X. In the initial state s_0 of P(X), s_0.versions = $\{T_X\}$, s_0.start = s_0.end = $\{<T_X,-\infty>\}$, and s_0.state = $\{<T_X,r_0>\}$, where r_0 is a start state of X. All other components of s_0 are empty.

In a state s, the components s.created, s.aborted, and s.committed keep track of those transactions for which the object has received a CREATE, INFORM-ABORT, and INFORM-COMMIT, respectively. The component s.commit-requested records all transactions for which the object has sent out a REQUEST-COMMIT. The component s.start records the start of each pseudotime range assigned to accesses of X by ASSIGN-PSEUDOTIME operations.

The remaining components of the state of P(X) constitute the multi-version history which P(X) uses to represent the states of X at various points in pseudotime. The component s.versions holds those writer accesses which have requested to commit and which are not orphaned in s. The component s.state maps each access T to a state of X which follows some schedule equieffective to the prefix of rearrange(generic(α),P'_α) ending in a REQUEST-COMMIT for T.

It is convenient to treat accesses as occurring at specific points in pseudotime; because ASSIGN-PSEUDOTIME assigns a range to an access, we obtain a point by choosing the beginning of the access's assigned range.[4] The principle which underlies the operation of P(X) is then straightforward. The versions, their states, and their associated ranges describe a partial history of an execution of X, viewed as taking place in pseudotime rather than real time. If s is a state of P(X), the fact that a particular version T is in s.versions represents the idea that the state of X during the interval [s.start(T),s.end(T)) would be equivalent to s.state(T), in the sense that it follows a schedule equieffective to some schedule which would leave X in s.state(T).

[4] This choice is, in fact, completely arbitrary; any point in the assigned range will do.

Regions of *P* which are not covered by intervals [s.start(T),s.end(T)) represent those regions in which no particular state must hold; thus writer accesses (which must change the state of X) can safely occur in those regions. When a reader access occurs in an unmarked region, we presume that the state of X at the pseudotime of the reader is the state of the previous writer access, and set the range of that writer access to extend from the pseudotime of the writer to the pseudotime of the reader, and thereby guarantee that no subsequent writer access will invalidate our presumption. When a version is initially created, its range is empty; and it is only through reference by readers that its range is extended. This process of transforming one partial history into another is described more fully in [R], under the name *eduction.*

In describing the pseudotime object algorithm, it will be useful to use some shorthand. Given a state s and a pseudotime $p \in P$, we define latest(s,p) to be that version in s.versions with the greatest value of s.start less than p. If T_1 and T_2 are accesses of X, we say T_1 is *visible in s* to T_2 if either $T_1 = T_X$, or every ancestor of T_1 which is a proper descendant of lca(T_1,T_2) is in s.committed. We say a transaction T is *orphaned in s* if any ancestor of T is in s.aborted.

We can now state the transition relation for P(X). The transitions of P(X) are exactly those tuples (s',π,s) satisfying the following pre- and postconditions.

- CREATE(T)
  Postcondition:
  s.created = s'.created ∪ {T}
- ASSIGN-PSEUDOTIME(T,P)
  Postcondition:
  s.start = s'.start ∪ {<T,P_{min}>}
  s.end = s'.end ∪ {<T,P_{min}>}
- INFORM-ABORT-AT(X)OF(T)
  Postcondition:
  s.aborted = s'.aborted ∪ {T}
  s.versions = {T' ∈ s'.versions | T' ∉ descendants(T) }
- INFORM-COMMIT-AT(X)OF(T)
  Postcondition:
  s.committed = s.committed ∪ {T}

- REQUEST-COMMIT(T,v), T a write
  Preconditions:
  T ∈ s'.created - s'.commit-requested
  T ∈ domain(s'.start)
  s'.aborted ∩ ancestors(T) = ∅
  If T' = latest(s',s'.start(T)), then
    s'.end(T') < s'.start(T)
  There exist states r_1, r_2 of X such that
    (r_0,CREATE(T),r_1) and
    (r_1,REQUEST-COMMIT(T,v),r_2) are both steps of X
  Postconditions:
  s.commit-requested = s'.commit-requested ∪ {T}
  s.versions = s'.versions ∪ {T}
  s.state = s'.state ∪ {<T,r_2>}

- REQUEST-COMMIT(T,v), T a read
  Preconditions:
  T ∈ s'.created - s'.commit-requested
  T ∈ domain(s'.start)
  s'.aborted ∩ ancestors(T) = ∅
  If T' = latest(s',s'.start(T)), then
    there exist states r_1, r_2 of X such that
      (s'.state(T'),CREATE(T),r_1) and
      (r_1,REQUEST-COMMIT(T,v),r_2) are both steps of X
    T' is visible to T in s'
  Postconditions:
  s.commit-requested = s'.commit-requested ∪ {T}
  s.end(T') = max(s'.end(T'),s.start(T))

- REQUEST-COMMIT(T,v), T an update
  Preconditions:
  T ∈ s'.created - s'.commit-requested
  T ∈ domain(s'.start)
  s'.aborted ∩ ancestors(T) = ∅
  If T' = latest(s',s'.start(T)), then
    s'.end(T') < s'.start(T)
    there exist states r_1, r_2 of X such that
      (s'.state(T'),CREATE(T),r_1) and
      (r_1,REQUEST-COMMIT(T,v),r_2) are both steps of X
    T' is visible to T in s'
  Postconditions:
  s.commit-requested = s'.commit-requested ∪ {T}
  s.versions = s'.versions ∪ {T}
  s.state = s'.state ∪ {<T,r_2>}
  s.end(T') = max(s'.end(T'),s'.start(T))

The pre- and postconditions of REQUEST-COMMIT do most of the real work. The postconditions maintain the partial history described by the versions, state, start, and end components; the preconditions are responsible for guaranteeing that no writers occur in a marked range, that the value returned by any access is that which would be returned in the execution of X described in the partial history, and that the writer which created the latest version at the pseudotime of a reader is always visible to that reader. The precondition on REQUEST-COMMIT that prevents a REQUEST-COMMIT for any access which is already orphaned at P(X), together with the second postcondition on INFORM-ABORT, ensures that no access appears in versions if it is orphaned at X.

The remaining pre- and postconditions of P(X) simply allow P(X) to record the occurrence of its operations, as described in the following lemma.

Lemma 4.15: Let α be a schedule of P(X) which can lead to a state s from the initial state. Then all of the following conditions hold.

1. T $\in$ s.created if and only if CREATE(T) appears in α.
2. T $\in$ s.aborted if and only if INFORM-ABORT-AT(X)OF(T) appears in α.
3. T $\in$ s.committed if and only if INFORM-COMMIT-AT(X)OF(T) appears in α.
4. T $\in$ s.commit-requested if and only if a REQUEST-COMMIT for T appears in α.
5. <T,p> $\in$ s.start if and only if ASSIGN-PSEUDOTIME(T,P) appears in α for some pseudotime range P such that $P_{min} = p$.

Proof: By induction on α using the postconditions of P(X).

The preceding lemma allows us to easily prove that P(X) preserves well-formedness.

Lemma 4.16: Let αREQUEST-COMMIT(T,v) be a schedule of P(X). Then the following conditions hold:

1. CREATE(T) appears in α.
2. ASSIGN-PSEUDOTIME(T,P) appears in α for some pseudotime range P.
3. REQUEST-COMMIT(T,v') does not appear in α for any value v'.
4. INFORM-ABORT-AT(X)OF(T') does not appear in α for any value v'.

Proof: By Lemma 4.15 and the preconditions on REQUEST-COMMIT(T,v).

Corollary 4.17: P(X) preserves well-formedness.

The following lemma describes the relationship between *visible in s* and *visible at X.*

Lemma 4.18: Let α be a schedule of P(X) which can lead to a state s from the initial state. Then if $T_1 \neq T_X$, T_1 is visible to T_2 in s if and only if T_1 is visible to T_2 at X in α.

Proof: From Lemma 4.15 and the definitions of the two notions of visibility.

We now turn our attention to versions and its associated components.

Lemma 4.19: Let α be a schedule of P(X) which can lead to a state s from the initial state; let T be an element of accesses(X) $\cup \{T_X\}$. Then $T \in$ s.versions if and only if $T = T_X$, or T is a writer and a REQUEST-COMMIT for T appears in done(α).

Proof: Suppose $T = T_X$; then $T \in s_0$.versions. Furthermore, $T \notin$ ancestors(T') for any T' for which P(X) might receive an INFORM-ABORT. Thus $T \in$ s.versions.

Alternatively, $T \neq T_X$. We consider three cases:

1. No REQUEST-COMMIT for T appears in α. Because $T \neq T_X$, $T \notin s_0$.versions. Then a simple induction on α using the postconditions on P(X) proves that $T \notin$ s.versions. Since no REQUEST-COMMIT for T appears in α, no REQUEST-COMMIT for T appears in done(α), and the condition holds.

2. An INFORM-ABORT for an ancestor of T appears in α. By Lemma 4.16, if any REQUEST-COMMIT for T appears in α, it precedes all INFORM-ABORTs for ancestors of T. Thus there are sequences β, γ such that $\alpha = \beta\gamma$, β ends in an INFORM-ABORT operation π for an ancestor of T, and γ contains no REQUEST-COMMITs for T. If s' is the state of P(X) after β, the postcondition on π requires that $T \notin$ s'.versions. By the method of the previous case, T cannot then be in s.versions. Since π appears in α, no operation mentioning T appears in done(α). Thus the condition holds.

3. A REQUEST-COMMIT for T appears in α, and no INFORM-ABORT appears in α for any ancestor of T. Then a REQUEST-COMMIT for T appears in done(α). By a simple induction on α using the postconditions of P(X), T is in s.versions, so the condition holds.

The states of P(X) satisfy certain consistency properties, described in the following lemma, which we will need in our correctness proof.

Lemma 4.20: Let α be a well-formed schedule of P(X) which leads to a state s from the initial state, and let α' be a prefix of α which leads to a state s'. Then all of the following are true:

1. s'.start $\subseteq$ s.start.

2. If $T \in$ domain(s'.end), $T \in$ domain(s.end) and s.end(T) $\geq$ s'.end(T).

3. If $T \in$ domain(s'.state), $T \in$ domain(s.state) and s.state(T) = s'.state(T).

Proof: By induction on α using the postconditions of P(X).

Reader accesses produce a host of ugly complications. To minimize the resulting confusion, we define the following notion of the access read by a particular reader.

Definition 4.21: Let α be a well-formed schedule of P(X), and let T be a reader such that a REQUEST-COMMIT operation π for T appears in α. Let α' be the longest prefix of α which does not contain a REQUEST-COMMIT for T, and let s' be the state of P(X) after α'. Then reads(T,α) = latest(s',s'.start(T)).

Fortunately, when α is a schedule of P(X) such that T is mentioned in done(α), we do not actually have to reconstruct the earlier state of P(X) to find reads(T,α), as we demonstrate in the following two lemmas.

Lemma 4.22: Let α be a well-formed schedule of P(X) which can lead to a state s from the initial state. Let T be an access in s.versions. Then if T = reads(α,T') for some reader T', s.end(T) $\geq$ s.start(T').

Proof: Let s' be the state following the REQUEST-COMMIT for T; by the postconditions on REQUEST-COMMIT, s'.end(T') $\geq$ s'.start(T). Now, using Lemma 4.20, we know that s'.start(T) = s.start(T) and s.end(T') $\geq$ s'.end(T'). Thus s.end(T') $\geq$ s'.end(T') $\geq$ s.start(T).

Lemma 4.23: Let α be a well-formed schedule of P(X) which can lead to a state s from the initial state. Let T be a reader mentioned in done(α). Then reads(T,α) = latest(s,s.start(T)).

Proof: Let α' be the longest prefix of α not containing a REQUEST-COMMIT for T; let s' be the state of P(X) following α'. Let T' = reads(T,α).

We begin by showing T' $\in$ s.versions. If T = T_X, then T' $\in$ s.versions by lemma 4.19. Otherwise, by the preconditions on REQUEST-COMMIT for reads and updates, T' is visible to T in s'. But then by Lemma 4.18 T' is visible to T at X in α', so T' is committed at X to lca(T,T') in α'. Now if T' is not in s.versions, an INFORM-ABORT must appear in α for some ancestor U of T'. By well-formedness U cannot be a proper descendant of lca(T,T'), so U is an ancestor of lca(T,T'). But then T' is an orphan at X in α, and thus cannot be mentioned in done(α), a contradiction.

Now suppose that T' $\in$ s.versions, but T' $\neq$ latest(s,s.start(T)). Then there must exist some other writer U in s.versions such that s.start(T') < s.start(U) < s.start(T). U cannot be T_X; thus a REQUEST-COMMIT π for U appears in α. If π appears in α', then U $\in$ s'.versions and s'.start(T') < s'.start(U) < s'.start(T), which contradicts T' = reads(T,α). Suppose instead, then, that there is some writer U such that a REQUEST-COMMIT for U occurs after a REQUEST-COMMIT for T, and s.start(T') < s.start(U) < s.start(T). Assume without loss of generality that U is the writer with the least value of s.start(U) which meets the condition. Let s'' be the state of P(X) following the REQUEST-COMMIT for U. Then T' = latest(s'',s''.start(U)). Now, by Lemma 4.22, s''.end(T') $\geq$

s''.start(T'); but then s''.end(T') $\geq$ s''.start(U), which contradicts the preconditions for REQUEST-COMMIT. Thus there can be no such U, and T' = latest(s,s.start(T)).

We now have sufficient resources to prove that P(X) meets its correctness condition.

Lemma 4.24: Let α be a well-formed schedule of P(X) which leads to a state s. Let β be an even-length prefix of rearrange(generic(α),P'_α), and let U be the last writer mentioned in β, or T_X if no writer is mentioned in β. Then there exists a well-formed schedule γ of X, which is a subsequence of β equieffective to β, and which leads to the state s.state(U) from the start state of X.

Proof: We proceed by induction on the length of β. If β is the empty sequence, then U = T_X, and γ is also the empty sequence. Otherwise, β = β'CREATE(T)REQUEST-COMMIT(T,v) for some β', T, and v. By induction hypothesis, the Lemma holds for β'; let U' and γ' be the appropriate access and schedule for β'. We note that both β and β', as prefixes of rearrange(generic(α),P'_α), must be well-formed. There are then three cases, depending on T:

- If T is a read access, then U = U' = latest(s,s.start(T)) and $\gamma = \gamma'$. By Lemma 4.23, U = reads(T,α) = latest(s',s'.start(T)) where s' is the state of P(X) immediately preceding REQUEST-COMMIT(T,v) in α. By Lemma 4.20, s'.state(U) = s.state(U). The preconditions on REQUEST-COMMIT(T,v) thus ensure that γ'CREATE(T)REQUEST-COMMIT(T,v) is a schedule of X; furthermore, by well-formedness of β neither CREATE(T) nor REQUEST-COMMIT(T,v'), for any v', can occur in γ', so γ'CREATE(T)REQUEST-COMMIT(T,v) is well-formed. Thus because γ' and β' are equieffective, by lemma 4.12 γ'CREATE(T)REQUEST-COMMIT(T,v) and β are equieffective.

- If T is a write access, then U = T. Let γ = CREATE(T)REQUEST-COMMIT(T,v). We know that β = β'CREATE(T)REQUEST-COMMIT(T,v) is well-formed; thus by Definition 4.14 β is equieffective to γ. That γ leads to s.state(U) follows immediately from the pre- and postconditions on REQUEST-COMMIT(T,v).

- If T is an update access, then U = T. Let γ = γ'CREATE(T)REQUEST-COMMIT(T,v). By Lemma 4.23, U' = reads(T,α) = latest(s',s'.start(T)) where s' is the state of P(X) immediately preceding REQUEST-COMMIT(T,v) in α. By Lemma 4.20, s'.state(U') = s.state(U'). The preconditions on REQUEST-COMMIT(T,v) thus ensure that γ = γ'CREATE(T)REQUEST-COMMIT(T,v) is a schedule of X; and as above, it is well-formed. Since β' and γ' are equieffective, by Lemma 4.12 β and γ are equieffective. Finally, the postconditions on REQUEST-COMMIT(T,v) and Lemma 4.20 guarantee that s.state(U) can follow from γ, where s'' is the state of P(X) immediately following REQUEST-COMMIT(T,v) in α.

Corollary 4.25: If α is a well-formed schedule of P(X), then rearrange(generic(α),P'_α) is a well-formed schedule of X.

Proof: By the lemma, there exists a well-formed schedule γ of X which is equieffective to rearrange(generic(α),P'_{α}). Since rearrange(generic(α),P'_{α}) is well-formed, it must also be a schedule of X.

4.3 Pseudotime Schedules

The pseudotime system is the composition of the pseudotime controller, the transactions of the serial system, and a pseudotime object P(X) for each basic object X in the serial system. A *pseudotime schedule* is simply a schedule of the pseudotime system. A sequence of operations of the pseudotime system is well-formed provided its projection on transactions and pseudotime objects is well-formed.

Lemma 4.26: Let α be a schedule of the pseudotime system. Then α is well-formed.

Proof: If α is the empty sequence, the α is well-formed. Otherwise, let $\alpha = \alpha'\pi$, where π is a single operation, and assume by induction hypothesis that α' is well-formed. There are several cases.

- π is an output of a transaction or pseudotime object. Then α is well-formed by the requirement that transactions and pseudotime objects preserve well-formedness.
- π is an input to a transaction T. Because α' is well-formed for the pseudotime system, we know, by Lemma 4.11 and the fact that generic(α')|T = α'|T for all transactions T, that generic(α') is well-formed for the generic system. Now, all operations of T are serial, so generic(α) = generic(α')π. By Lemma 4.1, generic(α) is a schedule of the generic controller; by Lemma 3.9, generic(α)|T = α|T is well-formed. Thus α is well-formed.
- π is an input to a pseudotime object P(X). If π is a REQUEST-COMMIT, INFORM-ABORT, or INFORM-COMMIT operation, then well-formedness of α|P(X) follows from Lemmas 4.1, 3.9, and 4.11. Otherwise, either π is CREATE(T) for some T or π is ASSIGN-PSEUDOTIME(T,P) for some T, P. In the former case, by Lemmas 4.1 and 3.2 CREATE(T) does not appear in α'; and by 4.5 ASSIGN-PSEUDOTIME(T,P) appears in α for some T, P; thus α is well-formed. In the latter case, 4.5 guarantees that at most one ASSIGN-PSEUDOTIME for T appears in α; thus no ASSIGN-PSEUDOTIME for T appears in α', and α is well-formed.

4.4 Serial Correctness

We can now prove that schedules of the pseudotime system are serially correct for non-orphans. First, we must show that, for any pseudotime schedule α and object P(X), P^*_α timestamp-orders generic(α)|P(X).

Lemma 4.27: Let α be a pseudotime schedule and P(X) an object. Then P^*_α timestamp-orders generic(α|P(X)).

Proof: By Lemma 4.26 α|P(X) is well-formed for P(X); by Lemma 4.11 generic(α|P(X)) is then well-formed for G(X). Now, if β is any sequence of INFORM-ABORT operations such that generic(α|P(X))β is well-formed for G(X), then (α|P(X))β will be well-formed for P(X). Furthermore, (α|P(X))β will be a schedule of P(X), since β contains only input operations of P(X) which must always be enabled. Thus by Corollary 4.25, rearrange(generic(α|P(X))β,P'_α) is a schedule of X. Now, P'_α is a subset of P^*_α by Lemma 4.9. Thus rearrange(generic(α|P(X))β,P^*_α) is a schedule of X for all sequences β of INFORM-ABORT operations such that generic(α|P(X))β is well-formed. Thus P^*_α timestamp-orders generic(α|P(X)).

Theorem 4.28: Let α be a pseudotime schedule. Then α is serially correct for non-orphans.

Proof: By Lemmas 4.1, 4.26, and 4.11, generic(α) is a generic schedule. By Lemma 4.7, P_α is a sibling order consistent with affects$_T$(α); we may then extend $P_\alpha \cup$ affects$_T$(α) into a total sibling order Q. Now, by the preceding lemma, P^*_α (and thus Q^*) timestamp-orders α|G(X) for every generic object G(X). Thus we can apply Theorem 3.31, and generic(α) is serially correct for non-orphans. Since generic(α)|T = α|T for any transaction T, α is thus also serially correct for non-orphans.

Chapter 5

Conclusions

We have defined, by means of Theorem 3.31, precise correctness properties for timestamp ordering algorithms, and provided a rigorous proof of the correctness of Reed's object history mechanism. In the process, we have defined a method for extracting the "visible" subsequence of a schedule of a non-serial system which refines the transaction-based notion of *visibility* of [LM, FLMW] to the level of individual operations. Our results have been stated in terms of a generic system which closely resembles the generic system of [FLMW, HLMW]; we hope that this will allow later work to combine timestamp ordering with other concurrency control algorithms in the same system, while retaining serial correctness.

Chapter 6

Acknowledgments

I would like to thank Nancy Lynch for her continued assistance and support throughout the course of this project. I would also like to extend my thanks to Alan Fekete for his comments on many drafts of this paper; and to Michael Merritt for providing both the idea of using a sibling order to represent timestamp ordering and, in a slightly different form from that used here, the proof of Lemma 3.18.

References

[BHG] Bernstein, P.A., Hadzilacos, V., and Goodman, N.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley, 1986.

[FLMW] Fekete, A., Lynch, N., Merritt, M., and Weihl, W.
Nested Transactions and Read/Write Locking.
In *Proceedings of the Symposium on Principles of Database Systems.* 1987.
To appear.

[HLMW] Herlihy, M., Lynch, N., Merritt, M., and Weihl, W.
On the Correctness of Orphan Elimination Algorithms.
Submitted for publication.

[LM] Lynch, N., and Merritt, M.
Introduction to the Theory of Nested Transactions.
Technical Report MIT/LCS/TR-367, MIT Laboratory for Computer Science, Cambridge, MA, July, 1986.

[R] Reed, D.P.
Naming and Synchronization in a Decentralized Computer System.
Technical Report MIT/LCS/TR-205, MIT Laboratory for Computer Science, Cambridge, MA, 1978.