A Theory of Timestamp-Based Concurrency Control for Nested Transactions

James Aspnes, C.M.U.
Alan Fekete, M.I.T.
Nancy Lynch, M.I.T.
Michael Merritt, AT&T Bell Labs
William Weihl, M.I.T.

Abstract:
We present a rigorous framework for analyzing timestamp-based concurrency control and recovery algorithms for nested transactions. We define a local correctness property, *local static atomicity*, that affords useful modularity. We show that local static atomicity of each object is sufficient to ensure global serializability. We present generalizations of algorithms due to Reed and Herlihy, and show that each ensures local static atomicity.

1 Introduction

Atomic transactions have been widely accepted as a mechanism for coping with concurrency and failures in database systems. In recent work, researchers have explored using transactions in more general distributed systems. *Nested transactions* [8, 7, 2, 9] have proved particularly useful in distributed systems, since they provide a simple mechanism for obtaining concurrency within a transaction, and for limiting the effects of a failure to a small part of a transaction.

In a recent series of papers, we have given a definition of correctness for nested transaction systems, and several algorithms for concurrency control, replication and orphan management have been described and verified. We refer to the reader to [4] for an account of the fundamental definitions, which give a framework for describing and verifying concurrency control and recovery algorithms for nested transactions.[2] In addition, [4] presents a Serializability Theorem in which a general correctness condition is shown to be sufficient to ensure serializability. This condition is based on the observation that the essential problem in ensuring correctness is to guarantee that there is a consistent serialization order at all objects in the system.

Different algorithms achieve the goal of a consistent serialization order in different ways. In this paper, we apply the general framework to algorithms in which timestamps are assigned to transactions when they begin execution, and in which transactions are serialized in timestamp order. We consider distributed implementations in which the state of the system is divided among a number of independent objects, and derive a local condition on individual objects, called *local static atomicity*, that ensures global correctness. This local condition provides us with modularity: we can use different concurrency control and recovery algorithms at different objects, as long as each object ensures local static atomicity. For example, in this paper we analyze two different timestamp-based concurrency control algorithms, both of which ensure local static atomicity. Our results imply that one of the algorithms could be used at some objects in a system, and the other at other objects, and global correctness would still be guaranteed. Such modularity is essential in distributed or object-oriented databases, in which different objects are implemented independently.

We give formal descriptions and correctness proofs for two algorithms: Reed's multi-version timestamp-based algorithm [8], and a generalization to nested transactions of Herlihy's type-specific timestamp-based algorithm [1], which was designed for single-level transactions. The latter algorithm uses the semantics of operations to permit a higher level of concurrency than can be achieved by treating the operations simply as reads and writes. In both cases, we describe the algorithms in a general fashion that permits a high level of concurrency and requires relatively few aborts by maintaining precise information about what transpires during an execution.

2 The Input/Output Automaton Model

The following is a brief introduction to the formal model that we use to describe and reason about systems. This model is treated in detail in [5] and [4].

All components in our systems, transactions, objects and

[1]The work of the first author was supported by a National Science Foundation fellowship. The work of the second and third authors was supported in part by the office of Naval Research under Contract N00014-85-K-0168, by the National Science Foundation under Grant CCR-8611442, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125. The work of the fifth author was supported in part by the National Science Foundation under Grant CCR-8716884, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

[2]The framework is based on the model of nested transaction systems originally presented in [3], but generalizes it to encompass a wider variety of algorithms.

schedulers, will be modelled by *I/O automata*. An I/O automaton A has a set of *states*, some of which are designated as *initial states*. It has *actions*, divided into *input actions*, *output actions* and *internal actions*. We refer to both input and output actions as *external actions*. It has a transition relation, which is a set of triples of the form (s',π,s), where s' and s are states, and π is an action. This triple means that in state s', the automaton can atomically do action π and change to state s. An element of the transition relation is called a *step* of the automaton.[3]

The input actions model actions that are triggered by the environment of the automaton, while the output actions model the actions that are triggered by the automaton itself and are potentially observable by the environment, and internal actions model changes of state that are not directly detected by the environment.

Given a state s' and an action π, we say that π is *enabled* in s' if there is a state s for which (s',π,s) is a step. We require that each input action π be enabled in each state s', i.e. that an I/O automaton must be prepared to receive any input action at any time.

A *finite execution fragment* of A is a finite alternating sequence $s_0\pi_1s_1\pi_2...\pi_ns_n$ of states and actions of A ending with a state, such that each triple (s',π,s) that occurs as a consecutive subsequence is a step of A. We also say in this case that $(s_0,\pi_1...\pi_n,s_n)$ is an *extended step* of A, and that (s_0,β,s_n) is a *move* of A where β is the subsequence of $\pi_1...\pi_n$ consisting of external actions of A. A *finite execution* is a finite execution fragment that begins with a start state of A.

From any execution, we can extract the *schedule*, which is the subsequence of the execution consisting of actions only. Because transitions to different states may have the same actions, different executions may have the same schedule. From any execution or schedule, we can extract the *behavior*, which is the subsequence consisting of the external actions of A. We write finbehs(A) for the set of all behaviors of finite executions of A.

We say that a finite schedule or behavior β *can leave* A *in* state s if there is some execution with schedule or behavior α and final state s. We say that an action π is *enabled after* a schedule or behavior α, if there exists a state s such that π is enabled in s and α can leave A in state s.

Since the same action may occur several times in an execution, schedule or behavior, we refer to a single occurrence of an action as an *event*.

We describe systems as consisting of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata, also. Thus, we define a composition operation for I/O automata, to yield a new I/O automaton. A collection of I/O automata is said to be *strongly compatible* if any internal action of any one automaton is not an action of any other automaton in the collection, any output action of one is not an output action of any other, and no action is shared by infinitely many automata in the collection. A collection of strongly compatible automata may be composed to create a *system* $\mathcal{S}$.

A state of the composed automaton is a tuple of states, one for each component automaton, and the start states are tuples consisting of start states of the components. An action of the composed automaton is an action of a subset of the component automata. It is an output of the system if it is an output for any component. It is an internal action of the system if it is an internal action of any component. During an action π of $\mathcal{S}$, each of the components that has action π carries out the action, while the remainder stay in the same state. If β is a sequence of actions of a system with component A, then we denote by $\beta|A$ the subsequence of β containing all the actions of A. Clearly, if β is a finite behavior of the system then $\beta|A$ is a finite behavior of A.

[3]Also, an I/O automaton has an equivalence relation on the set of output and internal actions. This is needed only to discuss fairness and will not be mentioned further in this paper.

3 Serial Systems and Correctness

In this section of the paper we summarize the definitions for serial systems, which consist of transaction automata and serial object automata communicating with a serial scheduler automaton. More details can be found in [4].

Transaction automata represent code written by application programmers in a suitable programming language. Serial object automata serve as specifications for permissible behavior of data objects. They describe the responses the objects should make to arbitrary sequences of operation invocations, assuming that later invocations wait for responses to previous invocations. The serial scheduler handles the communication among the transactions and serial objects, and thereby controls the order in which the transactions can take steps. It ensures that no two sibling transactions are active concurrently — that is, it runs each set of sibling transactions serially. The serial scheduler is also responsible for deciding if a transaction commits or aborts. The serial scheduler can permit a transaction to abort only if its parent has requested its creation, but it has not actually been created. Thus, in a serial system, all sets of sibling transactions are run serially, and in such a way that no aborted transaction ever performs any steps.

A serial system would not be an interesting transaction-processing system to implement. It allows no concurrency among sibling transactions, and has only a very limited ability to cope with transaction failures. However, we are not proposing serial systems as interesting implementations; rather, we use them exclusively as specifications for correct behavior of other, more interesting systems.

We represent the pattern of transaction nesting, a *system type*, by a set $\mathcal{T}$ of transaction names, organized into a tree by the mapping *parent*, with T_0 as the root. 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 this tree are called *accesses*. The accesses are partitioned so that each element of the partition contains the accesses to a particular object. In addition, the system type specifies a set of *return values* for transactions. If T is a transaction name that is an access to the object name X, and v is a return value, we say that the pair (T,v) is an *operation* of X.

The tree structure can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In any particular execution, however, only some of these transactions will actually take steps. We imagine that the tree structure is known in advance by all components of a system. The tree will, in general, be infinite and have infinite branching.

The classical transactions of concurrency control theory (without nesting) appear in our model as the children of a "mythical" transaction, T_0, the root of the transaction tree. Transaction T_0 models the environment in which the rest of the transaction system runs. It has actions that describe the invocation and return of the classical transactions. It is often natural to reason about T_0 in the same way as about all of the other transactions. The only transactions that actually access data are the leaves of the transaction tree, and thus they are distinguished as "accesses". (Note that leaves may exist at any level of the tree below the root.) The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly.

A serial system of a given system type is the composition of a set of I/O automata. This set contains a transaction automaton for each non-access node of the transaction tree, a serial object automaton for each object name, and a serial scheduler. These automata are described below.

3.1 Transactions

A non-access transaction T is modelled as a *transaction automaton* A(T), an I/O automaton with the following external actions. (In addition, A(T) may have arbitrary internal actions.)

Input:
  CREATE(T)
  REPORT_COMMIT(T',v), for T' a child of T,
    v a return value
  REPORT_ABORT(T'), for T' a child of T
Output:
  REQUEST_CREATE(T'), for T' a child of T
  REQUEST_COMMIT(T,v), for v a return value

The CREATE input action "wakes up" the transaction. The REQUEST_CREATE output action is a request by T to create a particular child transaction.[4] The REPORT_COMMIT input action reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The REPORT_ABORT input action reports to T the unsuccessful completion of one of its children, without returning any other information. The REQUEST_COMMIT action is an announcement by T that it has finished its work, and includes a value recording the results of that work.

We leave the executions of particular transaction automata largely unconstrained; the choice of which children to create and what value to return will depend on the particular implementation. For the purposes of the schedulers studied here, the transactions are "black boxes." Nevertheless, it is convenient to assume that behaviors of transaction automata obey certain syntactic constraints, for example that they do not request the creation of children before they have been created themselves and that they do not request to commit before receiving reports about all the children whose creation they requested. We therefore require that all transaction automata preserve *transaction well-formedness*, as defined formally in [4].

[4]Note that there is no provision for T to pass information to its child in this request. In a programming language, T might be permitted to pass parameter values to a subtransaction. Although this may be a convenient descriptive aid, it is not necessary to include in it the underlying formal model. Instead, we consider transactions that have different input parameters to be different transactions.

3.2 Serial Objects

Recall that transaction automata are associated with non-access transactions only, and that access transactions model abstract operations on shared data objects. We associate a single I/O automaton with each object name. The external actions for each object are just the CREATE and REQUEST_COMMIT actions for all the corresponding access transactions. Although we give these actions the same kinds of names as the actions of non-access transactions, it is helpful to think of the actions of access transactions in other terms also: a CREATE corresponds to an invocation of an operation on the object, while a REQUEST_COMMIT corresponds to a response by the object to an invocation. Thus, we model the serial specification of an object X (describing its activity in the absence of concurrency and failures) by a *serial object automaton* S(X) with the following external actions. (In addition S(X) may have arbitrary internal actions.)

Input:
  CREATE(T), for T an access to X
Output:
  REQUEST_COMMIT(T,v), for T an access to X

As with transactions, while specific objects are left largely unconstrained, it is convenient to require that behaviors of serial objects satisfy certain syntactic conditions. Let α be a sequence of external actions of S(X). We say that α is *serial object well-formed* for X if it is a prefix of a sequence of the form CREATE(T_1) REQUEST_COMMIT(T_1,v_1) CREATE(T_2) REQUEST_COMMIT(T_2,v_2) ..., where $T_i \neq T_j$ when $i \neq j$. We require that every serial object automaton preserve serial object well-formedness.[5]

3.3 Serial Scheduler

The third kind of component in a serial system is the serial scheduler. The transactions and serial objects have been specified to be any I/O automata whose actions and behavior satisfy simple restrictions. The serial scheduler, however, is a fully specified automaton, particular to each system type. It runs transactions according to a depth-first traversal of the transaction tree. The serial scheduler can choose nondeterministically to abort any transaction whose parent has requested its creation, as long as the transaction has not actually been created. Each child of T whose creation is requested must be either aborted or run to commitment with no siblings overlapping its execution, before T can commit. The result of a transaction can be reported to its parent at any time after the commit or abort has occurred.

The actions of the serial scheduler are as follows.

Input:
  REQUEST_CREATE(T), $T \neq T_0$
  REQUEST_COMMIT(T,v)
Output:
  CREATE(T)
  COMMIT(T), $T \neq T_0$
  ABORT(T), $T \neq T_0$
  REPORT_COMMIT(T,v), $T \neq T_0$
  REPORT_ABORT(T), $T \neq T_0$

[5]This is formally defined in [4] and means that the object does not violate well-formedness unless its environment has done so first.

The REQUEST_CREATE and REQUEST_COMMIT inputs are intended to be identified with the corresponding outputs of transaction and serial object automata, and correspondingly for the CREATE, REPORT_COMMIT and REPORT_ABORT output actions. The COMMIT and ABORT output actions mark the point in time where the decision on the fate of the transaction is irrevocable.

The details of the states and transition relation for the serial scheduler can be found in [4].

3.4 Serial Systems and Serial Behaviors

A *serial system* is the composition of a strongly compatible set of automata consisting of a transaction automaton A(T) for each non-access transaction name T, a serial object automaton S(X) for each object name X, and the serial scheduler automaton for the given system type.

The discussion in the remainder of this paper assumes an arbitrary but fixed system type and serial system, with A(T) as the non-access transaction automata, and S(X) as the serial object automata. We use the term *serial behaviors* for the system's behaviors. We give the name *serial actions* to the external actions of the serial system. The COMMIT(T) and ABORT(T) actions are called *completion* actions for T.

We introduce some notation that will be useful later. Let T be any transaction name. If π is one of the serial actions CREATE(T), REQUEST_CREATE(T'), REPORT_COMMIT(T',v'), REPORT_ABORT(T'), or REQUEST_COMMIT(T,v), where T' is a child of T, then we define *transaction*(π) to be T. If π is any serial action, then we define *hightransaction*(π) to be transaction(π) if π is not a completion action, and to be T, if π is a completion action for a child of T. Also, if π is any serial action, we define *lowtransaction*(π) to be transaction(π) if π is not a completion action, and to be T, if π is a completion action for T. If π is a serial action of the form CREATE(T) or REQUEST_COMMIT(T,v), where T is an access to X, then we define *object*(π) to be X.

If β is a sequence[6] of actions, T a transaction name and X an object name, we define $\beta|T$ to be the subsequence of β consisting of those serial actions π such that transaction(π) = T, and we define $\beta|X$ to be the subsequence of β consisting of those serial actions π such that object(π) = X. We define *serial*(β) to be the subsequence of β consisting of serial actions.

If β is a sequence of actions and T is a transaction name, we say T is an *orphan* in β if there is an ABORT(U) action in β for some ancestor U of T.

3.5 Serial Correctness

Following [3] we use the serial system to specify the correctness condition that we expect other, more efficient systems to satisfy. We say that a sequence β of actions is *serially correct* for transaction name T provided that there is some serial behavior γ such that $\beta|T = \gamma|T$. We will be interested primarily in showing, for particular systems of automata, representing a controller that generates timestamps and data objects that use different methods of concurrency control, that all finite behaviors are serially correct for T_0. As a sufficient condition, or as a stronger correctness condition of interest in its own right, we will show that all finite behaviors are serially correct for all non-orphan non-access transaction names. (Serial correctness for T_0 follows because the serial scheduler does not have an action ABORT(T_0).)

[6]We make these definitions for arbitrary sequences of actions, because we will use them later for behaviors of systems other than the serial system.

We believe serial correctness to be a natural notion of correctness that corresponds precisely to the intuition of how nested transaction systems ought to behave. Serial correctness for T is a condition that guarantees to implementors of T that their code will encounter only situations that can arise in serial executions. Correctness for T_0 is a special case that guarantees that the external world will encounter only situations that can arise in serial executions.

4 Simple Systems and the Serializability Theorem

In this section we outline a general method for proving that a concurrency control algorithm guarantees serial correctness. This method is treated in more detail in [4], and is an extension to nested transaction systems of ideas presented in [10, 11]. These ideas give formal structure to the simple intuition that a behavior of the system will be serially correct so long as there is a way to order the transactions so that when the operations of each object are arranged in that order, the result is legal for the serial specification of that object's type. For nested transaction systems, the corresponding result is Theorem 1. Later in this paper we will see that the essence of a nested transaction system using timestamps is that the serialization order is defined by the generated timestamps.

It is desirable to state our Serializability Theorem in such a way that it can be used for proving correctness of many different kinds of transaction-processing systems, with radically different architectures. We therefore define a "simple system", which embodies the common features of most transaction-processing systems, independent of their concurrency control and recovery algorithms, and even of their division into modules to handle different aspects of transaction-processing.

Many complicated transaction-processing algorithms can be understood as implementations of the simple system. For example, we will see that a system containing separate objects that manage multiple timestamped versions and a "controller" that generates timestamps and passes information among transactions and objects can be represented in this way.

4.1 Simple Database

There is a single simple database for each system type. The actions of the simple database are those of the composition of the serial scheduler with the serial objects:

Input:
- REQUEST_CREATE(T), $T \neq T_0$
- REQUEST_COMMIT(T,v), T a non-access

Output:
- CREATE(T)
- COMMIT(T), $T \neq T_0$
- ABORT(T), $T \neq T_0$
- REPORT_COMMIT(T,v), $T \neq T_0$
- REPORT_ABORT(T), $T \neq T_0$
- REQUEST_COMMIT(T,v), T an access

Each state s of the simple database consists of six sets, denoted via record notation: s.create_requested, s.created, s.commit_requested, s.committed, s.aborted and s.reported. The set s.commit_requested is a set of operations. The others are sets of transactions. There is exactly one start state, in which the set create_requested is $\{T_0\}$, and the other sets are empty. We use the notation s.completed to denote s.committed $\cup$ s.aborted. The transition relation is described below. (In describing transition relations, we use the convention that s' is the state of the automaton before the indicated action, and s is the target state. We explicitly describe only those components of the state s which are different from the components of s': thus, since the component s.created is not mentioned in the effect of the REQUEST_CREATE(T) action, it is implied that s.created = s'.created.)

REQUEST_CREATE(T), $T \neq T_0$
Effect:
  s.create_requested = s'.create_requested $\cup$ {T}

REQUEST_COMMIT(T,v), T a non-access
Effect:
  s.commit_requested = s'.commit_requested $\cup$ {(T,v)}

CREATE(T)
Precondition:
  T $\in$ s'.create_requested - s'.created
Effect:
  s.created = s'.created $\cup$ {T}

COMMIT(T), $T \neq T_0$
Precondition:
  (T,v) $\in$ s'.commit_requested for some v
  T $\notin$ s'.completed
Effect:
  s.committed = s'.committed $\cup$ {T}

ABORT(T), $T \neq T_0$
Precondition:
  T $\in$ s'.create_requested - s'.completed
Effect:
  s.aborted = s'.aborted $\cup$ {T}

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition:
  T $\in$ s'.committed
  (T,v) $\in$ s'.commit_requested
  T $\notin$ s'.reported
Effect:
  s.reported = s'.reported $\cup$ {T}

REPORT_ABORT(T), $T \neq T_0$
Precondition:
  T $\in$ s'.aborted
  T $\notin$ s'.reported
Effect:
  s.reported = s'.reported $\cup$ {T}

REQUEST_COMMIT(T,v), T an access
Precondition:
  T $\in$ s'.created
  for all v', (T,v') $\notin$ s'.commit_requested
Effect:
  s.commit_requested = s'.commit_requested $\cup$ {(T,v)}

Thus, the simple database embodies those constraints that we would expect any reasonable transaction-processing system to satisfy. It does not allow CREATEs, ABORTs, or COMMITs without an appropriate preceding request, does not allow any transaction to have two creation or completion events, and does not report completion events that never happened. Also, it does not produce responses to accesses that were not invoked, nor does it produce multiple responses to accesses. On the other hand, the simple database allows almost any ordering of transactions, allows concurrent execution of sibling transactions, and allows arbitrary responses to accesses. We do not claim that the simple database produces only serially correct behaviors; rather, we use the simple database to model features common to more sophisticated systems that do ensure correctness.

4.2 Simple Systems and Simple Behaviors

A *simple system* is the composition of a strongly compatible set of automata consisting of a transaction automaton A(T) for each non-access transaction name T, and the simple database automaton for the given system type. When the particular simple system is understood from context, we will use the term *simple behaviors* for the system's behaviors.

4.3 The Serializability Theorem

The type of transaction ordering needed for our theorem is more complicated than that used in the classical theory, because of the nesting involved here. Instead of just arbitrary total orderings on transactions, we will use partial orderings that only relate siblings in the transaction nesting tree. Formally, a *sibling order* R is an irreflexive partial order on transaction names such that (T,T') $\in$ R implies parent(T) = parent(T').

A sibling order R can be extended in two natural ways. First, R_{trans} is the binary relation on transaction names containing (T,T') exactly when there exist transaction names U and U' such that T and T' are descendants of U and U' respectively, and (U,U') $\in$ R. Second, if β is any sequence of actions, then $R_{event}(\beta)$ is the binary relation on events in β containing (ϕ,π) exactly when ϕ and π are distinct serial events in β with lowtransactions T and T' respectively, where (T,T') $\in R_{trans}$. It is clear that R_{trans} and $R_{event}(\beta)$ are irreflexive partial orders.

In order to state the Serializability Theorem we must introduce some technical definitions. Motivation for these can be found in [4].

First, we define when one transaction is "visible" to another. This captures a conservative approximation to the conditions under which the activity of the first can influence the second. Let β be any sequence of actions. If T and T' are transaction names, we say that T' is *visible* to T in β if there is a COMMIT(U) action in β for every U in ancestors(T') - ancestors(T). Thus, every ancestor of T' up to (but not necessarily including) the least common ancestor of T and T' has committed in β. If β is any sequence of actions and T is a transaction name, then *visible*(β,T) denotes the subsequence of β consisting of serial actions π with hightransaction(π) visible to T in β.

We define an "affects" relation. This captures basic dependencies between events. For a sequence β of actions, and events ϕ and π in β, we say that $(\phi,\pi) \in$ *directly-affects*(β) if at least one of the following is true: transaction(ϕ) = transaction(π) and ϕ precedes π in β,[7] ϕ = REQUEST_CREATE(T) and π = CREATE(T), ϕ = REQUEST_COMMIT(T,v) and π = COMMIT(T), ϕ = REQUEST_CREATE(T) and π = ABORT(T), ϕ = COMMIT(T) and π = REPORT_COMMIT(T,v), or ϕ = ABORT(T) and π = REPORT_ABORT(T). For a sequence β of actions, define the relation *affects*(β) to be the transitive closure of the relation directly-affects(β).

The following technical property is needed for the proof of Theorem 1. Let β be a sequence of actions and T a transaction name. A sibling order R is *suitable* for β and T if the following conditions are met.

1. R orders all pairs of siblings T' and T'' that are lowtransactions of actions in visible(β,T).
2. $R_{event}(\beta)$ and affects(β) are consistent partial orders on the events in visible(β,T).

We introduce some terms for describing sequences of operations. For any operation (T,v) of an object X, let *perform*(T,v) denote the sequence of actions CREATE(T)REQUEST_COMMIT(T,v). This definition is extended to sequences of operations: if $\xi=\xi'$(T,v) then perform(ξ)=perform(ξ')perform(T,v). A sequence ξ of operations of X is *serial object well-formed* if no two operations in ξ have the same transaction name. Thus if ξ is a serial object well-formed sequence of operations of X, then perform(ξ) is a serial object well-formed sequence of actions of X. We say that an operation (T,v) *occurs* in a sequence β of actions if a REQUEST_COMMIT(T,v) action occurs in β. Thus, any serial object well-formed sequence β of external actions of S(X) is either perform(ξ) or perform(ξ)CREATE(T) for some access T, where ξ is a sequence consisting of the operations that occur in β.

Finally we can define the "view" of a transaction at an object, according to a sibling order in a behavior. This is the fundamental sequence of actions considered in the hypothesis of the Serializabilty Theorem. Suppose β is a finite simple behavior, T a transaction name, R a sibling order that is suitable for β and T, and X an object name. Let ξ be the sequence consisting of those operations occurring in β whose transaction components are accesses to X and that are visible to T in β, ordered according to R_{trans} on the transaction components. (The first condition in the definition of suitability implies that this ordering is uniquely determined.) Define *view*(β,T,R,X) to be perform(ξ).

Theorem 1: (Serializability Theorem [4])
Let β be a finite simple behavior, T a transaction name such that T is not an orphan in β, and R a sibling order suitable for β and T. Suppose that for each object name X, view(β,T,R,X) $\in$ finbehs(S(X)). Then β is serially correct for T.

5 Timestamps

The essential feature of systems using timestamps is the explicit construction of a sibling order representing the intended serialization of an execution. This order is represented in terms of intervals of *pseudotime*, an arbitrarily-chosen totally-ordered set. Formally, we let $\mathcal{P}$ be the set of pseudotimes, ordered by <. We represent pseudotime intervals as half-open intervals [p,q) in $\mathcal{P}$. If P = [p,q), then we write P_{min} for p and P_{max} for q. If P and Q are intervals of pseudotime, we write $P < Q$ if $P_{max} \le Q_{min}$. Clearly, if $P < Q$, then P and Q are disjoint.

We model a database management system in which independently implemented objects respond to accesses using the timestamps assigned to the accesses. Formally, we consider a *distributed pseudotime system* in which the simple database is implemented with a separate *pseudotime object automaton* for each object name (this handles the concurrency control and data for that object in an as yet unspecified way, using information about timestamps of accesses and the fates of transactions) together with a *pseudotime controller* (this assigns timestamps and passes requests and information around the system.)

5.1 Pseudotime Objects

For each object name X, we define a *pseudotime object automaton* for X to be an automaton with the following external actions (the internal actions are unspecified) that preserve the well-formedness condition given below. We use P(X) to denote an arbitrary pseudotime object automaton for X.

Input:
 CREATE(T), T an access to X
 INFORM-COMMIT-AT(X)OF(T)
 INFORM-ABORT-AT(X)OF(T)
 INFORM_TIME_AT(X)OF(T,p), T $\in$ accesses(X), p $\in \mathcal{P}$
Output:
 REQUEST_COMMIT(T,v), T $\in$ accesses(X)

The actions CREATE(T) and REQUEST_COMMIT(T,v) correspond to invocations of, and responses to, accesses, just as for a serial object S(X). In addition, a pseudotime object has input actions that correspond to the receipt of information about the commit or abort of a transaction, and about the pseudotime assigned to an access transaction.

A sequence β of external actions of P(X) is *pseudotime object well-formed* for X if it satisfies the following conditions: no access is created more than once, no conflicting information is received about the fate of any transaction or the timestamp of any access, information that an access to X committed is received only after a REQUEST_COMMIT for the access, at most one REQUEST_COMMIT appears for any access, and any REQUEST_COMMIT for an access is preceded by both a CREATE and an INFORM_TIME_AT_X for the access. We require pseudotime objects to preserve pseudotime object well-formedness, that is, they may not produce an output action that causes the behavior to violate the conditions, unless the previous behavior had already violated the conditions (because of an input action.)

[7]This includes accesses as well as non-accesses.

5.2 Pseudotime Controller

The pseudotime controller has the following actions.

Input:
  REQUEST_CREATE(T), $T \neq T_0$
  REQUEST_COMMIT(T,v)
Output:
  ASSIGN_PSEUDOTIME(T,P), P a pseudotime interval, $T \neq T_0$
  CREATE(T)
  COMMIT(T), $T \neq T_0$
  ABORT(T), $T \neq T_0$
  REPORT_COMMIT(T,v), $T \neq T_0$
  REPORT_ABORT(T), $T \neq T_0$
  INFORM_COMMIT-AT(X)OF(T), $T \neq T_0$
  INFORM_ABORT-AT(X)OF(T), $T \neq T_0$
  INFORM_TIME_AT(X)OF(T,p), T an access to X, $p \in \mathcal{P}$

A state s of the pseudotime controller has the same components as a state of the simple database together with an additional component s.interval, which is a partial function from $\mathcal{T}$ to the set of pseudotime intervals. In the initial state of the pseudotime controller, interval = $\{(T_0,P_0)\}$, for some pseudotime interval P_0; and all other components are as in the initial state of the simple database. The transition relation is the same as that for the simple database, except as follows. The actions CREATE(T) and ABORT(T) have an additional precondition: $T \in$ domain(s'.interval). The action REQUEST_COMMIT(T,v) for T an access (which is an output of the simple database but an input of the pseudotime controller) has no precondition, and the additional actions are defined as follows.

```
ASSIGN_PSEUDOTIME(T,P)
Precondition:
  $T \in$ s'.create-requested
  $T \notin$ domain(s'.interval)
  $P \subseteq$ s'.interval(parent(T))
  $P >$ s'.interval(T'),
     for all $T' \in$ siblings(T) $\cap$ domain(s'.interval)
Effect:
  s.interval = s'.interval $\cup$ $\{(T,P)\}$

INFORM_COMMIT_AT(X)OF(T)
Precondition:
  $T \in$ s'.committed

INFORM_ABORT_AT(X)OF(T)
Precondition:
  $T \in$ s'.aborted

INFORM_TIME_AT(X)OF(T,p)
Precondition:
  $(T,P) \in$ s'.interval
  $p = P_{min}$
```

Thus the pseudotime controller assigns pseudotime intervals to transactions once the transactions creation has been requested. A transaction may not be created or aborted until it has been assigned a pseudotime interval. The pseudotime controller guarantees that siblings are assigned disjoint intervals of pseudotime within the interval assigned to their parent. In addition the order in pseudotime of the pseudotime intervals assigned to siblings is the same as the order during the execution in which those intervals were assigned. The pseudotime controller may also pass information to a pseudotime object about the completion of a transaction, or about the time assigned to an access to that object.

5.3 Distributed Pseudotime Systems

A *distributed pseudotime system* is the composition of a strongly compatible set of automata consisting of a transaction automaton A(T) for each non-access transaction name T, a pseudotime object automaton P(X) for each object name X, and the pseudotime controller for the given system type. When the particular distributed pseudotime system is understood from context, we use the term *distributed pseudotime behaviors* for the system's behaviors.

The following captures the relationship between a distributed pseudotime system and the simple system.

Lemma 2: If β is a distributed pseudotime behavior, then serial(β) is a simple behavior and serial(β)|T = β|T for all transaction names T.

Since the pseudotime controller and each pseudotime object automaton all preserve pseudotime object well-formedness, we have the following result.

Lemma 3: If β is a distributed pseudotime behavior and X is an object name, then β|P(X) is pseudotime object well-formed for X.

The purpose of the ASSIGN_PSEUDOTIME actions is to construct, at run-time, a sibling order that specifies the apparent serial ordering of transactions. If β is a sequence of events, then define the relation *pseudotime-order*(β) as follows. We say that $(T,T') \in$ pseudotime-order(β) exactly if T and T' are siblings, and ASSIGN_PSEUDOTIME(T,P) and ASSIGN_PSEUDOTIME(T',P') occur in β with $P < P'$.

One can prove the following result.

Lemma 4: Let β be a distributed pseudotime behavior and T a transaction name. Then pseudotime-order(β) is suitable for serial(β) and T.

We can thus apply our Serializability Theorem to obtain a sufficient condition for serial correctness of a distributed pseudotime behavior.

Lemma 5: Let β be a finite distributed pseudotime behavior, T a transaction name such that T is not an orphan in β, and let R = pseudotime-order(β). Suppose that for each object name X, view(serial(β),T,R,X) $\in$ finbehs(S(X)). Then β is serially correct for T.

Proof: Lemma 2 implies that serial(β) is a simple behavior and serial(β)|T = β|T. Since T is not an orphan in β, it is also not an orphan in serial(β). Lemma 4 implies that R is suitable for serial(β) and T. Then Lemma 1 implies that there is a finite serial behavior γ such that serial(β)|T = γ|T; thus β|T = γ|T, as needed. □

5.4 Static Atomicity

When designing a distributed database management system, we want our choice of concurrency control and recovery algorithms to guarantee the serial correctness of all the finite behaviors of the system. Also, we would like to be able to consider the algorithm used in implementing one object without reasoning about the implementations of all the other objects. Thus we are led to define the property of "static atomicity". This extends to nested transaction systems the local atomicity property of the same name defined in [10, 11].

Let P(X) be a pseudotime object automaton for object name X. We say that P(X) is *static atomic* for a given system type if for all distributed pseudotime systems $\mathcal{S}$ of the given type in which P(X) is associated with X, the following is true. Let β be a finite behavior of $\mathcal{S}$, R = pseudotime-order(β) and T a transaction name that is not an orphan in β. Then view(serial(β),T,R,X) ∈ finbehs(S(X)).

A distributed pseudotime system $\mathcal{S}$ is *static atomic* if every pseudotime object automaton in $\mathcal{S}$ is static atomic.

Theorem 6: A static atomic system is serially correct for every non-orphan transaction.

Proof: Immediate from Lemma 5. □

5.5 Local Static Atomicity

In this subsection, we provide a sufficient condition for showing that a pseudotime object automaton P(X) is static atomic. The condition, called "local static atomicity", depends only on the behaviors of P(X), and does not require reasoning about the context in which P(X) is placed. This condition generalizes work of Weihl for transaction systems without nesting [10, 11].

First we introduce some terms to describe information about the behavior of a distributed pseudotime system that pseudotime object automaton P(X) can deduce from its local behavior. Let β be a sequence of external actions of P(X), and let T and T' be transaction names. Then T is *locally visible* to T' in β if β contains an INFORM_COMMIT_AT(X)OF(U) for every U in ancestors(T) - ancestors(T'). Also, T is a *local orphan* in β if β contains an INFORM_ABORT_AT(X)OF(U) for some ancestor U of T. Also we define a binary relation *local-pseudotime-order*(β) on accesses to X, where (T,T') ∈ local-pseudotime-order(β) exactly when β contains both INFORM_TIME_AT(X)OF(T,p) and INFORM_TIME_AT(X)OF(T',p') events for some p and p' with $p < p'$.

Now suppose β is a pseudotime object well-formed sequence of external actions of pseudotime object automaton P(X). Let *local-view*(β,T) be the unique sequence defined as follows. Let S be the set of operations occurring in β whose transactions are locally visible to T in β. Then local-view(β,T) = perform(ξ), where ξ is the result of reordering S according to the order R = local-pseudotime-order(β) on the transaction components.

We say that pseudotime object automaton P(X) is *locally static atomic* if whenever β is a finite pseudotime object well-formed behavior of P(X), and T is a transaction name that is not a local orphan in β, then local-view(β,T) ∈ finbehs(S(X)).

We now justify the names introduced above by showing some relationships between the local properties defined above and the corresponding global properties.

Lemma 7: Let β be a behavior of a distributed pseudotime system in which pseudotime object automaton P(X) is associated with X. If transaction name U is a local orphan in β|P(X) then U is an orphan in β. Similarly if U is locally visible to V in β|P(X) then U is visible to V in β. If R = pseudotime-order(β) and R' = local-pseudotime-order(β|P(X)), then $R' \subseteq R_{trans}$ and $R'_{event}(\beta|P(X)) \subseteq R_{event}(\beta)$.

Proof: These are immediate consequences of the pseudotime controller preconditions, which imply that any INFORM_COMMIT_AT(X)OF(U) is preceded by COMMIT(U), and analogously for INFORM_ABORT and INFORM_TIME actions. □

Finally we show the main result of this subsection, that local static atomicity is a sufficient condition for static atomicity.

Theorem 8: If P(X) is a pseudotime object automaton that is locally static atomic then P(X) is static atomic.

Proof: Let $\mathcal{S}$ be a distributed pseudotime system in which P(X) is associated with X. Let β be a finite behavior of $\mathcal{S}$, R = pseudotime-order(β) and T a transaction name that is not an orphan in β. We must prove that δ = view(serial(β),T,R,X) ∈ finbehs(S(X)).

Let γ be a finite sequence of actions consisting of exactly one INFORM_COMMIT_AT(X)OF(U) for each COMMIT(U) that occurs in β. Then βγ is a behavior of the system $\mathcal{S}$, since each action in γ is an output of the pseudotime controller that is enabled. By Lemma 3, βγ|P(X) is a pseudotime object well-formed behavior of P(X).

Since INFORM_COMMIT_AT(X)OF(U) occurs in βγ|P(X) if and only if COMMIT(U) occurs in β, an access T' to X is visible to T in β if and only if it is locally visible to T in βγ|P(X). Since T is not an orphan in β, it is not an orphan in βγ, and thus not a local orphan in βγ|P(X). Also, R = pseudotime-order(β) = pseudotime-order(βγ). Moreover, Lemma 7 implies that if accesses are ordered by local-pseudotime-order(βγ|P(X)) they are also ordered in the same way by R_{trans}.

Thus view(serial(β),T,R,X) = local-view(βγ|P(X),T). Since P(X) is locally static atomic, local-view(βγ|P(X),T) is in finbehs(S(X)). □

6 Reed's Algorithm

In this section we describe a generalization of the multi-version timestamp-based algorithm proposed by Reed [8]. The algorithm works for objects whose serial specification is such that each access is either a read (which does not alter the object in any essential way) or a write (which essentially obliterates any previous state). The algorithm keeps for each write access the version of the serial specification object S(X) that resulted. For each read access the algorithm records the writer of the version that was used to compute the result returned. If the reader is not an orphan, the version it used must be the effective version for the interval of pseudotime from that assigned to the writer until that assigned to the reader. A write access can not occur if there is already an effective version at the pseudotime assigned to the writer. A read access must use the version written by the non-orphan write access whose pseudotime most closely precedes the time assigned to the reader; however the read must not return unless the writer involved is visible to it. Extending the algorithm to deal with more general updates is straightforward, as these accesses both use an old version and produce a new version.

We describe the algorithm formally and show that it gives a locally static atomic object, and then we discuss how the correctness of Reed's algorithm follows.

6.1 Semantics of Operations

In order to use the algorithm of this section to provide a concurrent object whose serial specification is given by S(X), we need to have a classification of the accesses to X into two sorts, *read accesses* and *write accesses*. The correctness of the algorithm depends on the fact that the operations performed by S(X) for those accesses have the appropriate semantics: in essence, read accesses do not alter the state while write accesses leave no trace of the previous state.

We introduce a technical definition of "equieffective" sequences of actions to express precisely when the two sequences leave the object in essentially the same state. Let X be an object name and S(X) a particular serial object automaton for X. Let α and β be finite sequences of external actions of S(X). Then α is *equieffective* to β with respect to S(X) if, for every sequence γ of external actions of S(X) such that both $\alpha\gamma$ and $\beta\gamma$ are serial object well-formed, $\alpha\gamma$ is a behavior of S(X) if and only if $\beta\gamma$ is a behavior of S(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 either α or β is not serial object well-formed, they are trivially equieffective. Similarly, if both α and β are serial object well-formed sequences of external actions of S(X) and neither is a behavior of S(X) then they are trivially equieffective. On the other hand, α and β are serial object well-formed sequences of external actions of S(X), α is equieffective to β, and β is a behavior of S(X), then α must also be a behavior of S(X).

Extensions of equieffective sequences are equieffective. This notion is stated more formally in the following lemma.

Lemma 9: Let X be an object name and S(X) a serial object for X. Let α and β be equieffective sequences of external actions of S(X). Let γ be a finite sequence of external actions of S(X). Then $\alpha\gamma$ is equieffective to $\beta\gamma$.

We use equieffectiveness to define the essential properties of read accesses. Namely, let T be an access to X. Then T is a *read access* if, for any sequence α of external actions of S(X) and value v such that αperform(T,v) is a serial object well-formed behavior of S(X), αperform(T,v) is equieffective to α. That is, a read access is one that cannot be detected by later accesses to X.

This definition has the following fundamental consequence which shows that two sequences of operations have essentially the same effect on an object provided that they differ only in the presense of a collection of read accesses.

Lemma 10: Let X be an object name and S(X) a serial object automaton for X. Let η be a sequence of operations of X such that perform(η) is a serial object well-formed behavior of S(X), and let ξ be a subsequence of η, such that every access in ξ is a read access. Then perform(η) and perform(η-ξ) are equieffective serial object well-formed behaviors of S(X).

Similarly, we use equieffectiveness to define the property of write accesses that is essential for the algorithms and proofs, i.e. that they prevent the detection of any preceding accesses. Let T be an access to X. Then T is a *write access* if, for any behavior α of S(X) and value v such that αperform(T,v) is serial object well-formed, αperform(T,v) is equieffective to perform(T,v).

6.2 Abstract Description

Suppose X is an object name and S(X) a serial object automaton for X. Suppose we are given a classification of the accesses to X into two classes, one of only read accesses and the other of only write accesses. Note that not every S(X) allows such a classification of accesses, which we call a *read-write classification.*

We now describe a particular pseudotime object automaton R(X) for X.

Each state s of R(X) has the following components: s.created, s.committed, s.aborted and s.commit-requested, which are sets of transactions, plus s.time, s.version and s.reads-from. Let T_X be a 'dummy' transaction name, used to represent the writer of the start state of S(X). The component s.time is a partial mapping from accesses(X) $\cup$ $\{T_X\}$ to $\mathcal{P} \cup \{-\infty\}$. The component s.version is a partial mapping from {T: T is a write accesses to X} $\cup$ $\{T_X\}$ to the set of states of the serial object automaton S(X), and the component s.reads-from is a partial mapping from {T: T is a write access to X} $\cup$ $\{T_X\}$ to the set of sets of read accesses to X.

In the initial state s_0 of R(X), s_0.time = $\{(T_X,-\infty)\}$, where we take $-\infty$ to be less than any pseudotime in $\mathcal{P}$. Also, s_0.version = $\{(T_X,r_0)\}$, where r_0 is a start state of X. Furthermore, s_0.reads-from = $\{(T_X,\emptyset)\}$. All other components of s_0 are empty.

In a state s, the components s.created, s.committed, and s.aborted keep track of those transactions for which the object has received a CREATE, INFORM_COMMIT and INFORM_ABORT, respectively. In addition, the component s.time keeps track of the pseudotime associated with each access transaction for which an INFORM_TIME has been received. The component s.commit-requested records all transactions for which the object has sent out a REQUEST_COMMIT. The s.version component keeps track of the versions of S(X) produced by different write accesses, while the s.reads-from component keeps track of the set of read accesses that read from a particular version.

In describing the pseudotime object algorithm, it will be useful to use some shorthand. Given a state s, we define s.orphans = {T : ancestors(T) $\cap$ s.aborted $\neq \emptyset$} and s.nonorphans = $\mathcal{T}$ - s.orphans. If T_1 and T_2 are accesses of X, we say T_1 is *visible in s* to T_2 if ancestors(T_1) - ancestors(T_2) $\subseteq$ s.committed.

The actions are as follows.

```
CREATE(T)
Effect:
   s.created = s'.created ∪ {T}

INFORM_COMMIT_AT(X)OF(T)
Effect:
   s.committed = s.committed ∪ {T}

INFORM_ABORT_AT(X)OF(T)
Effect:
   s.aborted = s'.aborted ∪ {T}

INFORM_TIME_AT(X)OF(T,p)
Effect:
   s.time(T) = p
   s.time(U) = s'.time(U) for U ≠ T
```

```
REQUEST_COMMIT(T,v), T a write access to X
Precondition:
    $T \in s'.created$
    $T \notin s'.commit\text{-}requested$
    $T \in domain(s'.time)$
    if $T'' \in s'.reads\text{-}from(T')$
        and $s'.time(T') < s'.time(T) < s'.time(T'')$
      then $T'' \in s'.orphans$
    $(r_0, perform(T,v), r)$ is a move of S(X)
Effect:
    $s.commit\text{-}requested = s'.commit\text{-}requested \cup \{T\}$
    $s.version(T) = r$
    $s.version(U) = s'.version(U)$ for $U \neq T$
    $s.reads\text{-}from(T) = \emptyset$
    $s.reads\text{-}from(U) = s'.reads\text{-}from(U)$ for $U \neq T$
```

```
REQUEST_COMMIT(T,v), T a read access to X
Precondition:
    $T \in s'.created$
    $T \notin s'.commit\text{-}requested$
    $T \in domain(s'.time)$
    $T' \in domain(s'.version)$
    $s'.time(T') < s'.time(T)$
    if $T'' \in domain(s'.version)$
        and $s'.time(T') < s'.time(T'') < s'.time(T)$
      then T'' is in s'.orphans
    either $T' = T_X$ or T' is visible to T in s'
    $(s'.version(T'), perform(T,v), r)$ is a move of S(X)
Effect:
    $s.commit\text{-}requested = s'.commit\text{-}requested \cup \{T\}$
    $s.reads\text{-}from(T') = s'.reads\text{-}from(T') \cup \{T\}$
    $s.reads\text{-}from(U) = s'.reads\text{-}from(U)$ for $U \neq T'$
```

The principle that underlies the action of R(X) is straightforward. The versions and reads-from relationships give partial information about an execution of S(X), viewed as taking place in pseudotime order rather than real time order. If s is a state of R(X), the facts that s.version(T') = r and T $\in$ s.reads-from(T') mean that (provided the accesses involved are not aborted) the state of S(X) during the interval [s.time(T'),s.time(T)) is equivalent to r in the sense that it follows a behavior equieffective to some behavior that would leave S(X) in state r.

Regions of P that are not covered by any interval [s.time(T'),s.time(T'')), for nonorphan write T' and nonorphan read T'' with T'' in s.reads-from(T') represent those regions in which no particular state must hold; thus write accesses can safely occur only in those regions. Thus a REQUEST_COMMIT(T,v) action for a write access has a precondition that checks that the pseudotime associated with the write does not lie in any interval [s.time(T'),s.time(T'')), for nonorphan write T' and nonorphan read T'' with T'' in s.reads-from(T'). Another precondition checks that the operation perform(T,v) can occur starting from r_0, which is an initial state of S(X), and an effect of the action is to record the resulting state of S(X) as s.version(T).

If T is a read access, it should return a value that is appropriate for the this access to S(X) occurring from the state of S(X) produced by the nonorphan writer whose timestamp is closest to (but preceding) the timestamp associated to T. The read access must be delayed until the writer is visible to T.

Next we give the basic invariants of the algorithm.

Lemma 11: Let β be a finite pseudotime object well-formed schedule of R(X). Suppose that β can lead to a state s from the initial state. Then the following conditions hold.

1. $T_X \in$ domain(s.version) and s.version(T_X) = r_0.
2. If T $\in$ domain(s.version) and T $\neq T_X$ then
 (a) T is a write access to X
 (b) there exists v such that REQUEST_COMMIT(T,v) occurs in β
 (c) T $\in$ domain(s.time), and
 (d) (r_0,perform(T,v),s.version(T)) is a move of S(X).
3. If T $\in$ s.reads-from(T'), then
 (a) T is a read access,
 (b) either T' = T_X or T' is visible to T in s,
 (c) T' and T are in domain(s.time) and s.time(T') < s.time(T)
 (d) if T'' $\in$ domain(s.version), T $\in$ s.nonorphans and s.time(T') < s.time(T'') < s.time(T) then T'' $\in$ s.orphans, and
 (e) there exist v and r such that
  (e1) REQUEST_COMMIT(T,v) occurs in β
  (e2) (s.version(T'),perform(T,v),r) is a move of S(X).
4. If T is in s.commit-requested then
 (a) if T is a write access then T $\in$ domain(s.version)
 (b) if T is a read access then there exists a T' such that T $\in$ s.reads-from(T').

Proof: By induction on the length of β. □

We now demonstrate that the algorithm above is satisfactory for use in a system that uses timestamp order for serialization, provided the accesses to S(X) have a read-write classification.

First we define a technical notion, capturing the intuition of a mutually consistent collection of non-orphan accesses. If s is a state of R(X), then define $\mathcal{U}$ to be an *allowable* set of accesses to X for s provided that the following conditions hold.

- All accesses in $\mathcal{U}$ are in s.nonorphans.
- All accesses in $\mathcal{U}$ are in s.commit-requested.
- $\mathcal{U}$ is closed under s.reads-from, i.e. if T' $\in$ s.reads-from(T) and T' $\in \mathcal{U}$, then either T = T_X or T $\in \mathcal{U}$.

The following lemma shows that the operations of an allowable set of accesses, arranged in timestamp order, provide a behavior of S(X). This is the key to the proof that the algorithm provides static atomicity.

Lemma 12: Let R(X) be constructed using a read-write classification of the accesses to X. Let β be any finite schedule of R(X) that is pseudotime object well-formed for X, and suppose β can lead to state s. Let $\mathcal{U}$ be an allowable set of accesses to X for s. Let S be the set of operations occurring in β whose transaction components are in $\mathcal{U}$. Let ξ be the result of ordering S using the order determined by s.time on the transaction components. Let γ = perform(ξ). Then $\gamma \in$ finbehs(S(X)).

Proof: By pseudotime object well-formedness, no transaction has more than one REQUEST_COMMIT in β, and thus all operations in S have distinct transaction components. Thus perform(ξ) is serial object well-formed.

We proceed by induction on the length of prefixes ζ of ξ, showing that for each such ζ, perform(ζ) is a behavior of S(X). The basis is trivial, so suppose that $\zeta = \zeta'$(T,v), where ζ is a prefix of ξ, perform(ζ') $\in$ finbehs(S(X)) and (T,v) is an operation in S. There are two cases.

1. T is a write access.
Then perform(ζ) = perform(ζ'(T,v)) is equieffective to perform(T,v). By the preconditions of REQUEST_COMMIT(T,v) in R(X), there is a move

(r_0,perform(T,v),r) of S(X). Thus, perform(T,v) is a behavior of S(X). Since perform(ζ) is equieffective to perform(T,v), perform(ζ) is also a behavior of S(X).

2. T is a read access.
Since $\mathcal{U}$ is allowable for s, T is in s.reads-from(T') for some T' which is either T_X or else is a write access in $\mathcal{U}$.
If T' = T_X then s.time(T') = $-\infty$, so by part 3 of Lemma 11, ζ' is a sequence of operations of the form (T'',v'') where each T'' is a read access. Then Lemma 10 implies that perform(ζ') is equieffective to the empty sequence. By Lemma 9, perform(ζ) = perform(ζ'(T,v)) is equieffective to perform(T,v). Since T $\in$ s.reads-from(T_X), Parts 1 and 3 of Lemma 11 imply that s.version(T_X) = r_0 and that there is a move (r_0,perform(T,v),r) of S(X). Thus, perform(T,v) $\in$ behs(S(X)). Since perform(ζ) is equieffective to perform(T,v), perform(ζ) $\in$ behs(S(X)).
If on the other hand T' $\neq$ T_X, then T' $\in$ $\mathcal{U}$. Then parts 2 and 3 of Lemma 11 imply that β contains a REQUEST_COMMIT(T',v') action, and s.time(T') < s.time(T). Thus, ζ' also contains the operation (T',v'), i.e. ζ' is of the form η(T',v')η'. By part 3 of Lemma 11, η' is a sequence of operations of the form (T'',v''), where each T'' is a read access. Then Lemma 10 implies that perform(ζ') = perform(η(T',v')η') is equieffective to perform(η(T',v')); this in turn is equieffective to perform(T',v'). Thus, perform(ζ') is equieffective to perform(T',v'). Then Lemma 9 implies that perform(ζ) is equieffective to perform(T',v')perform(T,v).
Since T $\in$ s.reads-from(T'), part 3 of Lemma 11 implies that there is a move (s.version(T'),perform(T,v),r) of S(X). Also, since T' $\in$ domain(s.version), part 2 of Lemma 11 implies that there is a move (r_0,perform(T',v'),s.version(T')) of S(X). Thus, perform(T',v')perform(T,v) is in behs(S(X)). Since perform(ζ) is equieffective to perform(T',v')perform(T,v), perform(ζ) $\in$ behs(S(X)).

□

Now we can prove that a data object is locally static atomic if it is implemented using the algorithm described in this section.

Theorem 13: Let R(X) be constructed using a read-write classification of the accesses to X. Then R(X) is locally static atomic.

Proof: Let β be a finite pseudotime object well-formed behavior of R(X), and T a transaction name that is not a local orphan in β. Let S be the set of operations occurring in β whose transactions are locally visible to T in β. Then local-view(β,T) = perform(ξ), where ξ is the result of ordering S according to the order R = local-pseudotime-order(β) on the transaction components. We must show that local-view(β,T) $\in$ finbehs(S(X)).

Since R(X) has no internal actions, β is also a schedule of R(X). Choose state s of R(X) such that β can lead to s. Let $\mathcal{U}$ be the set of transaction components of operations in S. We show that $\mathcal{U}$ is an allowable set of accesses to X for s.

Let T' be an access in $\mathcal{U}$. Since T is not a local orphan in β and T' is locally visible to T in β, we deduce that T' is not a local orphan in β. Thus, T' $\in$ s.nonorphans.

Since a REQUEST_COMMIT for T' occurs in β, T' $\in$ s.commit-requested.

Now suppose T' $\in$ s.reads-from(T'') and T'' $\neq$ T_X. Then part 3 of Lemma 11 implies that T'' is visible to T' in s, so that T'' is locally visible to T' in β. Thus, T'' is locally visible to T in β. Also by part 3 of Lemma 11, there is a REQUEST_COMMIT for T'' in β, i.e. T'' $\in$ $\mathcal{U}$. It follows that $\mathcal{U}$ is an allowable set of accesses to X, for s.

Now let S' be the set of operations occurring in β whose transaction components are in $\mathcal{U}$. Let ξ' be the result of ordering S' using s.time on the transaction components. Let γ' = perform(ξ'). Then Lemma 12 implies that γ' $\in$ finbehs(S(X)).

However, we observe that pseudotime object well-formedness implies that each access in $\mathcal{U}$ has at most one REQUEST_COMMIT in β. Thus, S = S'. Furthermore, local-pseudotime-order(β) is the same as the order determined by s.time. Therefore, ξ = ξ' and hence γ = γ'. Therefore, γ $\in$ finbehs(S(X)), as required. □

6.3 Reed's Implementation

The algorithm described by Reed differs from ours primarily in the way in which the reads-from component of the state is maintained. As described above, our algorithm maintains, for each version of the object written by a write access, the entire set of read accesses that read that version. Reed's algorithm maintains instead simply the maximum timestamp of all accesses that read a given version. It is easy to show that Reed's algorithm implements ours, in the sense that any behavior permitted by Reed's is permitted by ours. However, there are behaviors permitted by our algorithm that are not permitted by Reed's. For example, if a read access T reads a version written by T' and then aborts, our algorithm permits a later writer T'' whose timestamp falls between the timestamps for T' and T to create a new version. Reed's algorithm would force T'' to abort in this situation, since the maximum timestamp of readers for a given version is not decreased when a reader aborts. If aborts are rare, then Reed's implementation is probably adequate.

7 Type-specific Concurrency Control

The previous section described an algorithm that could be used for an object whose serial specification (its type) allows the accesses to be divided into read accesses and write accesses. In this section we give an algorithm that applies to any object and that uses more information about the semantics of the operations of the object. In fact, what we discuss here is a family of algorithms, one for each possible choice of dependency relation among the accesses. The algorithm is a generalization to encompass nesting of the work of [1]; in addition, we describe the preconditions on operations in a somewhat more general way than in [1], thus permitting more concurrency and avoiding some aborts.

Unlike Reed's algorithm described earlier, this algorithm does not keep versions of the serial object; instead it keeps track of all the operations that have occurred. The algorithm is based on the following idea: an access T is allowed to execute if all conflicting accesses with earlier timestamps are visible to T, and if T does not conflict with any accesses with later timestamps. The conflict relation must be a serial dependency relation, which we now define.

7.1 Serial Dependency Relations

Our definition of serial dependency relations is stated in a slightly different way from Herlihy's; however, it is easy to show that the two are equivalent. Examples of serial dependency relations can be found in [1]. We note in particular that every serial object S(X) has at least the trivial serial dependency relation that relates all operations of S(X).

To begin, let R be a binary relation on operations of serial object S(X). If ξ is a serial object well-formed sequence of operations of S(X) and η is a subsequence of ξ, then we say that η is *R-closed* in ξ provided that whenever η contains an operation π, it also contains all preceding operations ϕ of ξ such that $(\phi,\pi) \in R$.

Now, we say that R is a *serial dependency* relation for S(X) provided that the following holds, for all well-formed sequences ξ of operations of S(X). If for each π in ξ, there is an R-closed subsequence η of ξ containing π such that perform(η) is a behavior of S(X), then perform(ξ) is a behavior of S(X).

The following lemma gives the crucial properties needed to show that the algorithm described in the next section is correct.

Lemma 14: Suppose R is a serial dependency relation for S(X). Let η be a serial object well-formed sequence of operations of S(X) and let η' and η'' be subsequences of η such that each operation of η is in at least one of η' and η''. Suppose that perform(η') and perform(η'') are both behaviors of S(X). Moreover, suppose that the following hold.

1. If $\pi_1 \in \eta' - \eta''$ and $\pi_2 \in \eta'' - \eta'$ and π_1 precedes π_2 in η, then $(\pi_1,\pi_2) \notin R$.
2. If $\pi_1 \in \eta'' - \eta'$ and $\pi_2 \in \eta' - \eta''$ and π_1 precedes π_2 in η, then $(\pi_1,\pi_2) \notin R$.

Then perform(η) is a behavior of S(X).

Proof: Note that by the two numbered conditions, η' and η'' are R-closed subsequences of η. Since every operation π in η occurs either in η' or η'', it follows from the definition of serial dependency relation that perform(η) $\in$ behs(S(X)). □

7.2 Abstract Description

Let R be a serial dependency relation for S(X). We now describe a particular pseudotime object automaton H(X) for object name X.

Each state s of H(X) has the following components: s.created, s.committed, s.aborted, which are sets of transactions, s.commit-requested, which is a set of operations, and s.time, which is a partial mapping from accesses(X) to *P*. Initially, all sets are empty and s.time is undefined everywhere. As before, some shorthand is useful. Given a state s, we define s.orphans = $\{T : \text{ancestors}(T) \cap \text{s.aborted} \neq \emptyset\}$ and s.nonorphans = *T* - s.orphans. If T_1 and T_2 are accesses of X, we say T_1 is *visible in s* to T_2 if ancestors(T_1) - ancestors(T_2) $\subseteq$ s.committed. The actions are as follows.

```
CREATE(T)
Effect:
   s.created = s'.created ∪ {T}

INFORM_COMMIT_AT(X)OF(T)
Effect:
   s.committed = s'.committed ∪ {T}

INFORM_ABORT_AT(X)OF(T)
Effect:
   s.aborted = s'.aborted ∪ {T}

INFORM_TIME_AT(X)OF(T,p)
Effect:
   s.time(T) = p
   s.time(U) = s'.time(U) for U ≠ T

REQUEST_COMMIT(T,v), T an access to X
Precondition:
   T ∈ s'.created
   (T,w) ∉ s'.commit-requested for any w
   T ∈ domain(s'.time)
   if (T',v') ∈ s'.commit-requested, T' ∈ s'.nonorphans,
                         and ((T,v),(T',v')) ∈ R,
     then s'.time(T') < s'.time(T)
   let S = {(T',v') ∈ s'.commit-requested: T' ∈ s'.nonorphans
                         and s'.time(T') < s'.time(T)}
   if (T',v') ∈ S and ((T',v'),(T,v)) ∈ R,
     then T' is visible to T in s'
   let β be the result of ordering, in s'.time order,
     the operations (T',v') in S such that T' is visible to T in s'
   perform(β(T,v)) is a behavior of S(X)
Effect:
   s.commit-requested = s'.commit-requested ∪ {(T,v)}
```

The preconditions for REQUEST_COMMIT(T,v) deserve some explanation. The first three simply require that T has been created, its pseudotime is known to the object, and it has not requested to commit already. The next precondition requires that no nonorphan access with a later pseudotime depend on T; this is analogous to the "ratchet locks" of [1]. The fourth precondition requires that T not depend on nonorphan accesses with earlier pseudotimes that are not visible to T. The final precondition requires that the operation (T,v) be allowed by S(X) after the operations visible to T with earlier pseudotimes.

The next two lemmas give the main properties of this algorithm that we will use to prove its correctness. First we show that the conditions checked in the preconditions of a REQUEST_COMMIT(T,v) action remain true as long as T is not an orphan.

Lemma 15: Let β be a finite pseudotime object well-formed schedule of H(X). Suppose that β can lead to state s. Then the following conditions hold.

1. Suppose (T,v) $\in$ s.commit-requested and T $\in$ s.nonorphans. Let (T',v') $\in$ s.commit-requested, with T' $\in$ s.nonorphans, s.time(T') < s.time(T) and ((T',v'),(T,v)) $\in$ R. Then T' is visible to T in s.
2. Suppose (T,v) $\in$ s.commit-requested and T $\in$ s.nonorphans. Let S = {(T',v') $\in$ s.commit-requested: T' is visible to T in s and s.time(T') $\leq$ s.time(T)}. Let η be the result of ordering the operations in S in s.time order. Then perform(η) $\in$ behs(S(X)).

Proof: The proof is by induction on the length of β. The basis is obvious, so consider the inductive step. Let $\beta = \beta'\pi$, where π is a single action, where β' can lead to state s', and where (s',π,s) is a step of H(X).

Let (T,v) and (T',v') satisfy the hypothesis of Claim 1. Since T and T' are in s.nonorphans, they are in s'.nonorphans also. By well-formedness and the fact that T and T' are in s.commit-requested, it follows that T and T' are in domain(s'.time), and that s'.time(T) = s.time(T) and s'.time(T') = s.time(T'). There are three cases.

1. (T,v) and (T',v') are both in s'.commit-requested.
   Then the inductive hypothesis implies that T' is visible to T in s' and therefore in s.
2. (T,v) $\in$ s'.commit-requested and π is REQUEST_COMMIT(T',v').
   This contradicts the "ratchet lock" precondition of π.
3. (T',v') $\in$ s'.commit-requested and π is REQUEST_COMMIT(T,v).
   Then the precondition of π ensures that T' is visible to T in s' and hence in s.

Claim 1 follows.

Now let (T,v), S and η satisfy the hypothesis of Claim 2. Let S' = {(T',v') $\in$ s'.commit-requested: T' is visible to T in s' and s'.time(T') $\leq$ s'.time(T)}. Let η' be the result of ordering the operations in S' in s'.time order.

If π is REQUEST_COMMIT(T,v), then S = S' $\cup$ {(T,v)}, $\eta = \eta'$(T,v), and the operation sequence called β in the preconditions of π is equal to η'. Thus the claim is guaranteed by the preconditions of π. So assume that π is not REQUEST_COMMIT(T,v). Then (T,v) $\in$ s'.commit-requested. Since T $\in$ s'.nonorphans, the inductive hypothesis ensures that perform(η') is in behs(S(X)).

Now we consider cases.

1. π is neither a REQUEST_COMMIT(T',v') action nor an INFORM_COMMIT_AT(X)OF(U) for U a child of an ancestor of T.
   Then S' = S and $\eta' = \eta$, so the result follows.
2. π = REQUEST_COMMIT(T',v'), where T' $\neq$ T.
   Then pseudotime object well-formedness implies that S = S' and $\eta = \eta'$, so again the result follows. (T' cannot become visible to T as a result of this action, since INFORM_COMMIT_AT(X)OF(T') cannot precede REQUEST_COMMIT(T',v') in a pseudotime object well-formed sequence.)
3. π = INFORM_COMMIT_AT(X)OF(U) for U a child of an ancestor of T.
   Then consider the operations in S - S'. Let T'' be the access with the largest value of s'.time among those which appear as first component in S - S'. Clearly, (T'',v'') $\in$ s'.commit-requested for some v'' and T'' $\in$ s'.nonorphans. Let S'' = {(T',v') $\in$ s'.commit-requested: T' is visible to T'' in s' and s'.time(T') $\leq$ s'.time(T'')}. Let η'' be the result of ordering the operations in S'' in s'.time order. The inductive hypothesis ensures that perform(η'') is in behs(S(X)). Also note that S = S' $\cup$ S''; since T'' is visible to T in s, T'' is visible to U in s', and an access is visible to U in s' if and only if it is visible to T'' in s'. Thus, η' and η'' are both subsequences of η, and each operation in η is in at least one of the two subsequences. Claim 1 shows we can apply Lemma 14 to see perform(η) $\in$ behs(S(X)).

□

We now prove a property almost the same as local static atomicity, but stated using components of the object's state rather than properties of the behavior when determining which accesses are visible and non-orphans, and the order to use in arranging operations.

Lemma 16: Let β be a finite schedule of H(X) such that the behavior of β is pseudotime object well-formed. Suppose that β can lead to state s. Let T be in s.nonorphans. Let ξ be the result of ordering the operations (T',v') in s.commit-requested such that T' is visible to T in s in s.time order. Then perform(ξ) is a behavior of S(X).

Proof: Consider any operation (T',v') in ξ. By claim 2 of Lemma 15, perform(η) is a behavior of S(X), where η is the result of ordering the operations (T'',v'') $\in$ s.commit-requested with T'' visible to T' in s and s.time(T'') $\leq$ s.time(T'), in s.time order. Since (by Claim 1 of Lemma 15), η is an R-closed subsequence of ξ containing (T',v'), the fact that R is a serial dependency relation implies that perform(ξ) is a behavior of S(X). □

Now we can prove that a data object is locally static atomic if it is implemented using the algorithm described in this section.

Theorem 17: H(X) is locally static atomic.

Proof: Let β be a finite pseudotime object well-formed behavior of H(X), and T a transaction name that is not a local orphan in β. Let S be the set of operations occurring in β whose transactions are locally visible to T in β. Then local-view(β,T) = perform(ξ), where ξ is the result of ordering S according to the order R' = local-pseudotime-order(β) on the transaction components. We must show that local-view(β,T) $\in$ finbehs(S(X)).

Since H(X) has no internal actions, β is a schedule of H(X). Choose a state s of H(X) such that β can lead to s. Since T is not a local orphan in β, T is in s.nonorphans. Furthermore any transaction is locally visible to T in β if and only if it is visible to T in s, and the order defined by s.time is the same as R'.

Now the result is immediate from Lemma 16. □

Herlihy's algorithm was originally described for single-level transaction systems. We have extended it here to encompass nested transactions. In addition, we have generalized it slightly, by testing for conflicts on "ratchet locks" only for non-orphan transactions. Herlihy's algorithm does not release ratchet locks when a transaction aborts, so conflicts on ratchet locks are tested against all transactions, aborted or not. This is similar to the way Reed's algorithm maintains only the maximum timestamp for a reader of a given version, regardless of whether the readers have committed or aborted.

8 Conclusion

In this paper we have presented a rigorous framework for analyzing timestamp-based concurrency control and recovery algorithms for nested transactions. We defined a local correctness property, *local static atomicity*, that affords useful modularity, both in decomposing proofs and in building systems. The correctness proof is split into two parts: one showing that local static atomicity is sufficient to ensure global serializability, and another showing that particular algorithms ensure local static atomicity. The first part of the proof need be done only once; the second part must be done for each separate algorithm. In building systems, this decomposition allows us to use different algorithms at different objects, as long as each ensures local static atomicity.

Finally, we presented generalizations of algorithms due to Reed and Herlihy, and showed that each ensures local static atomicity. We extended Herlihy's algorithm to encompass nested transactions, and generalized both algorithms to handle information about aborted transactions more precisely. We are currently extending the algorithms and proof techniques to provide an integrated treatment of optimistic and pessimistic techniques, with each access making an independent decision whether to run optimistically or pessimistically.

The results of [4] can be used in a parallel manner to describe and verify *local dynamic atomic* algorithms, which use the order in which sibling transactions complete as the basis for serialization. Such algorithms include strict two-phase locking and extensions of it that use type-specific information to increase concurrency. We are currently writing a book [6] that will include all these results, as well as proofs of other algorithms used in nested transaction systems, such as those for replication management.

References

[1] Herlihy, M. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers* C-36(4), April, 1987.

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

[3] Lynch, N. A., and Merritt, M. *Introduction to the theory of nested transactions.* Technical Report MIT-LCS-TR-367, Massachusetts Institute of Technology, 1986. To appear in TCS.

[4] Lynch, N. A., Merritt, M., Weihl, W. E., and Fekete, A. A theory of atomic transactions. In *Proceedings of the International Conference on Database Theory.* 1988.

[5] Lynch, N., and Tuttle, M. *Hierarchical correctness proofs for distributed algorithms.* Technical Report MIT-LCS-TR-387, Massachusetts Institute of Technology, 1987. A shortened version appeared in PODC 87.

[6] Lynch, N. A., Merritt, M., Weihl, W. E., and Fekete, A. *Atomic Transactions.* In preparation.

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

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

[9] Spector, A., and Swedlow, K. (eds). *Guide to the Camelot distributed transaction facility: release 1.* Technical Report, Carnegie-Mellon University, 1988.

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

[11] Weihl, W. E. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems* , 1988. Accepted for publication.