Commutativity-Based Locking for Nested Transactions

Alan FEKETE, University of Sydney

Nancy LYNCH, Massachusetts Institute of Technology

Michael MERRITT, AT&T Bell Laboratories

William WEIHL, Massachusetts Institute of Technology

ABSTRACT:

We introduce a new algorithm for concurrency control in nested transaction systems. The algorithm uses semantic information about an object (commutativity of operations) to obtain more concurrency than is available with Moss' locking algorithm which is currently used as the default in systems like Argus and Camelot. We define "dynamic atomicity", a local property of an object, and prove that dynamic atomicity of each object guarantees the correctness of the whole system. Objects implemented using the commutativity-based locking algorithm are dynamic atomic.

October 5, 1989[1]

[1]This research was done while the first author was at Massachusetts Institute of Technology. The work of the first and second 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 fourth 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.

1 INTRODUCTION

The abstract notion of "atomic transaction" was originally developed to hide the effects of failures and concurrency in centralized database systems. Recently, a generalization to "nested transactions" has been advocated as a way to structure distributed systems in which information is maintained in persistent modifiable objects of abstract data types. Examples of systems using nested transactions are Argus [6] and Camelot [10]. In these systems "atomic" objects can be created, and operations on these objects will be serializable, and the state of the objects will survive failures of the nodes on which they reside. In both Argus and Camelot the default algorithm used for concurrency control is the locking protocol of Moss [9], but the implementor of an object has the option to write his or her own concurrency control and recovery routine. In this paper we introduce a general algorithm that uses semantic information (that is, the type of the object) to obtain more concurrency than is available with Moss' algorithm. Our algorithm is described in a very general form. Many detailed implementation choices can be made and the correctness of the resulting more specified implementation follows from the correctness of our general algorithm. Due to lack of space, all details of the correctness proof have been omitted from this proceedings, but they can be found in [3].

As is well known, errors can result if different concurrency control techniques are carelessly combined within a single system. We do more than prove our algorithm correct. We give a local condition called "dynamic atomicity" on an implementation of an object. We use our recently developed theory [7] to show that if each object in a system is dynamic atomic, then the whole system is serially correct. In [3] we show that our algorithm produces a dynamic atomic object, as does Moss' algorithm. Thus our algorithm can be used on a few concurrency bottlenecks in a system, while the simpler algorithm of Moss can be used elsewhere, without violating serial correctness.[2]

We have also used our theory elsewhere to present and prove correctness of several other kinds of transaction-processing algorithms, including timestamp-based algorithms for concurrency control and recovery [1] and algorithms for management of replicated data [4] and of orphan transactions [5].

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 [8] and [7].

All components in our systems, transactions, objects and 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*

[2] Our earlier paper [2] contains a direct proof of the serial correctness of systems where Moss' algorithm is used for every object.

actions. We refer to both input and output actions as *external actions.* We call the classification of actions the *action signature* of the automaton, and the classification with the internal actions omitted as the *external action signature.* We use the terms in(A), out(A), ext(A) to refer to the sets of input actions, output actions and external actions of the automaton A. An automaton 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 $s_0\pi_1 s_1\pi_2 \ldots \pi_n s_n$ 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 $(s_0,\pi_1 \ldots \pi_n,s_n)$ is an *extended step* of A, and that (s_0,β,s_n) is a *move* of A where β is the subsequence of $\pi_1 \ldots \pi_n$ 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

[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.

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.

Let A and B be automata with the same external actions. Then A is said to *implement* B if finbehs(A) $\subseteq$ finbehs(B). One way in which this notion can be used is the following. Suppose we can show that an automaton A is "correct", in the sense that its finite behaviors all satisfy some specified property. Then if another automaton B implements A, B is also correct.

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 [7].

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 [7].

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, v a return value

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)

[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.

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), for $T \neq T_0$
REQUEST_COMMIT(T,v) for T a transaction name, v a value
Output:
CREATE(T) for T a transaction name
COMMIT(T), for $T \neq T_0$
ABORT(T), for $T \neq T_0$
REPORT_COMMIT(T,v), for $T \neq T_0$, v a value
REPORT_ABORT(T), for $T \neq T_0$

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 [7].

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

[5]This is formally defined in [7] and means that the object does not violate well-formedness unless its environment has done so first.

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

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 data objects that use different methods of concurrency control and a controller that passes information between transactions and objects, 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).)

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 cor-

[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.

rectness 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 [7], and is an extension to nested transaction systems of ideas presented in [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 locking algorithms like Moss' is that the serialization order is defined by the order in which siblings complete.

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 locks and a "controller" that passes information among transactions and objects can be represented in this way.

We first define an automaton called the *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), for $T \neq T_0$
REQUEST_COMMIT(T,v), for T a non-access transaction name, v a value
Output:
CREATE(T) for T a transaction name
COMMIT(T), for $T \neq T_0$
ABORT(T), for $T \neq T_0$
REPORT_COMMIT(T,v), for $T \neq T_0$, v a value
REPORT_ABORT(T), for $T \neq T_0$
REQUEST_COMMIT(T,v), for T an access transaction name, v a value

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. The details can be found in [7]. 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.

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.

The Serializability Theorem is formulated in terms of simple behaviors; it provides a sufficient condition for a simple behavior to be serially correct for a particular transaction name T.

4.1 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 [7].

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[7])*
Let β be a finite simple behavior, T a transaction name such that T is not an orphan

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

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 DYNAMIC ATOMICITY

In this section, we specialize the ideas summarized in the preceding section to the particular case of locking algorithms. Locking algorithms serialize transactions according to a particular sibling order, the order in which transactions complete. Also, locking algorithms can be described naturally using a particular decomposition into a "generic object" automaton for each object name that handles the concurrency control and recovery for that object, and a single "generic controller" automaton that handles communication among the other components. We define the completion order and the appropriate system decomposition in this section.

We then give a variant of the Serializability Theorem, specialized for algorithms using the completion order and based on the given system decomposition. We call this theorem the Dynamic Atomicity Theorem, because it is stated in terms of a property of generic objects called "dynamic atomicity", which we also define in this section.

5.1 Completion Order

A key property of locking algorithms is that they serialize transactions according to their completion (commit or abort) order. This order is determined dynamically. If β is a sequence of events, then we define *completion*(β) to be the binary relation on transaction names containing (T,T') exactly if T and T' are siblings and one of the following holds.

1. There are completion events for both T and T' in β, and a completion event for T precedes a completion event for T'.
2. There is a completion event for T in β, but there is no completion event for T' in β.

The following is not hard to verify.

Lemma 2 *Let β be a finite simple behavior and T a transaction name. Then completion(β) is suitable for β and T.*

5.2 Generic Systems

In this subsection, we give the system decomposition appropriate for describing locking algorithms. We will formulate such algorithms as "generic systems", which are composed

of transaction automata, "generic object automata" and a "generic controller". The general structure of the system is the same as that for serial systems.

The object signature for a generic object contains more actions than that for serial objects. Unlike the serial object for X, the corresponding generic object is responsible for carrying out the concurrency control and recovery algorithms for X, for example by maintaining lock tables. In order to do this, the automaton requires information about the completion of some of the transactions, in particular, those that have visited that object. Thus, a generic object automaton has in its signature special INFORM_COMMIT and INFORM_ABORT input actions to inform it about the completion of (arbitrary) transactions.

5.2.1 Generic Object Automata

A *generic object automaton* G for an object name X of a given system type is an I/O automaton with the following external action signature.

Input:
CREATE(T), for T an access to X
INFORM_COMMIT_AT(X)OF(T), for T any transaction name
INFORM_ABORT_AT(X)OF(T), for T any transaction name
Output:
REQUEST_COMMIT(T,v), for T an access to X and v a value

In addition, G may have an arbitrary set of internal actions. G is required to preserve "generic object well-formedness", defined as follows. A sequence β of actions π in the external signature of G is said to be *generic object well-formed* for X provided that the following conditions hold.

1. There is at most one CREATE(T) event in β for any transaction T.
2. There is at most one REQUEST_COMMIT event in β for any transaction T.
3. If there is a REQUEST_COMMIT event for T in β, then there is a preceding CREATE(T) event in β.
4. There is no transaction T for which both an INFORM_COMMIT_AT(X)OF(T) event and an INFORM_ABORT_AT(X)OF(T) event occur.
5. If an INFORM_COMMIT_AT(X)OF(T) event occurs in β and T is an access to X, then there is a preceding REQUEST_COMMIT event for T.

5.2.2 Generic Controller

There is a single generic controller for each system type. It passes requests for the creation of subtransactions to the appropriate recipient, makes decisions about the commit or abort of transactions, passes reports about the completion of children back to their parents, and informs objects of the fate of transactions. Unlike the serial scheduler, it does not prevent sibling transactions from being active simultaneously, nor does it prevent the same transaction from being both created and aborted. Rather, it leaves the task of coping with concurrency and recovery to the generic objects.

The generic controller is a very nondeterministic automaton. It may delay passing requests or reports or making decisions for arbitrary lengths of time, and may decide at any time to abort a transaction whose creation has been requested (but that has not yet completed). Each specific implementation of a locking algorithm will make particular choices from among the many nondeterministic possibilities. For instance, Moss [9] devotes considerable effort to describing a particular distributed implementation of the controller that copes with node and communication failures yet still commits a subtransaction whenever possible. Our results apply *a fortiori* to all implementations of the generic controller obtained by restricting the nondeterminism.

The generic controller has the following action signature.

Input:
REQUEST_CREATE(T) for T a transaction name
REQUEST_COMMIT(T,v) for T a transaction name, v a value
Output:
CREATE(T) for T a transaction name
COMMIT(T), for $T \neq T_0$
ABORT(T), for $T \neq T_0$
REPORT_COMMIT(T,v), for $T \neq T_0$, v a value
REPORT_ABORT(T), for $T \neq T_0$
INFORM_COMMIT_AT(X)OF(T), for $T \neq T_0$
INFORM_ABORT_AT(X)OF(T), for $T \neq T_0$

All the actions except the INFORM actions play the same roles as in the serial scheduler. The INFORM_COMMIT and INFORM_ABORT actions pass information about the fate of transactions to the generic objects.

The transition relation for the generic controller is given in [3].

5.2.3 Generic Systems

A *generic system* of a given system type is the composition of a strongly compatible set of automata consisting of the transaction automaton A_T for each non-access transaction name T (this is the same automaton as in the serial system), a generic object automaton

G_X for each object name X, and the generic controller automaton for the system type.

The external actions of a generic system are called *generic actions*, and the executions, schedules and behaviors of a generic system are called *generic executions*, *generic schedules* and *generic behaviors*, respectively.

The following variant of the corollary to the Serializability Theorem applies to the special case where the sibling order is the completion order and the system is a generic system.

Proposition 3 *Let β be a finite generic behavior, T a transaction name that is not an orphan in β and R = completion(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.*

5.3 Dynamic Atomicity

Now we define the "dynamic atomicity" property for a generic object automaton; roughly speaking, it says that the object satisfies the view condition using the completion order as the sibling order R. This restatement of the view condition as a property of a generic object is very convenient for decomposing correctness proofs for locking algorithms: the Serializability Theorem implies that if all the generic objects in a generic system are dynamic atomic, then the system guarantees serial correctness for all non-orphan transaction names. All that remains is to show that the generic objects that model the locking algorithms of interest are dynamic atomic.

This proof structure can be used to yield much stronger results than just the correctness of the locking algorithm in this paper. As long as each object is dynamic atomic, the whole system will guarantee that any finite behavior is serially correct for all non-orphan transaction names. Thus, we are free to use an arbitrary implementation for each object, independent of the choice of implementation for each other object, as long as dynamic atomicity is satisfied. For example, a simple algorithm such as Moss's can be used for most objects, while a more sophisticated algorithm permitting extra concurrency by using type-specific information can be used for objects that are "hot spots". (That is, objects that are very frequently accessed.) The idea of a condition on objects that guarantees serial correctness was introduced by Weihl [11] for systems without transaction nesting.

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

Theorem 4 *(Dynamic Atomicity Theorem)*
Let $\mathcal{S}$ be a generic system in which all generic objects are dynamic atomic. Let β be a finite behavior of $\mathcal{S}$. Then β is serially correct for every non-orphan transaction name.

Proof: Immediate from Proposition 3 and the definition of dynamic atomicity. □

6 RESTRICTED TYPES OF SERIAL OBJECTS

The correctness of the algorithm in this paper depends on semantic information about the types of serial object automata used in the underlying serial system. In this section, we provide the appropriate definitions for these concepts.

We first define the important concept of "equieffectiveness" of two sequences of external actions of a serial object automaton. Roughly speaking, two sequences are "equieffective" if they can leave the automaton in states that are indistinguishable to the outside world. We then define the notion of "commutativity" required for our algorithm.

6.1 Equieffectiveness

Now we define "equieffectiveness" of finite sequences of external actions of a particular serial object automaton S_X. The definition says that the two sequences can leave S_X in states that cannot be distinguished by any environment in which S_X can appear. Formally, we express this indistinguishability by requiring that S_X can exhibit the same behaviors as continuations of the two given sequences.

Let X be an object name, and recall that S_X is a particular serial object automaton for X. Let β and β' be finite sequences of actions in ext(S_X). Then β is *equieffective* to β' if for every sequence γ of actions in ext(S_X) such that both $\beta\gamma$ and $\beta'\gamma$ are serial object well-formed, $\beta\gamma \in$ beh(S_X) if and only if $\beta'\gamma \in$ beh(S_X). Obviously, equieffectiveness is a symmetric relation, so that if β is equieffective to β' we often say that β and β' are *equieffective*. Also, any sequence that is not serial object well-formed is equieffective to all sequences. On the other hand, if β and β' serial object well-formed sequences and β is equieffective to β', then if β is in beh(S_X), β' must also be in beh(S_X).

A special case of equieffectiveness occurs when the final states of two finite executions are identical. The classical notion of serializability uses this special case, in requiring concurrent executions to leave the database in the same state as some serial execution of the same transactions. However, this property is probably too restrictive for reasoning about an implementation, in which details of the system state may be different following any concurrent execution than after a serial one. (Relations may be stored on different pages, or data structures such as B-trees may be configured differently.) Presumably, these details are irrelevent to the perceived future behavior of the database, which is an "abstraction" or "emergent property" of the implementation. The notion of equieffectiveness formalizes this indistinguishability of different implementation states.

6.2 Commutativity

We now define an appropriate notion of commutativity for operations of a particular serial object automaton.[8] Namely, we say that operations (T,v) and (T',v') *commute*, where T and T' are accesses to X, if for any sequence of operations ξ such that both perform(ξ(T,v)) and perform(ξ(T',v')) are serial object well-formed behaviors of S_X, then perform(ξ(T,v)(T',v')) and perform(ξ(T',v')(T,v)) are equieffective serial object well-formed behaviors of S_X.

Example: Consider an object S_X representing a bank account. The accesses to X are of the following kinds:

- balance?: The return value for this access gives the current balance.

  deposit_$a: This increases the balance by $a. The only return value is "OK".

- withdraw_$b: This reduces the balance by $b if the result will not be negative. In this case the return value is "OK". If the result of withdrawing would be to cause an overdraft, then the balance is left unchanged, and the return value is "FAIL".

For this object, it is clear that two serial object well-formed schedules that leave the same final balance in the account are equieffective, since the result of each access depends only on the current balance. We claim that if T and T' are accesses of kind deposit_$a and deposit_$b, then the operations (T,"OK") and (T',"OK") commute. To see this, suppose that perform(ξ(T,"OK")) and perform(ξ(T',"OK")) are serial object well-formed behaviors of S_X. This implies that ξ is serial object well-formed and contains no operation with first component T or T'. Therefore, β = perform(ξ(T,"OK")(T',"OK")) and β' = perform(ξ(T',"OK")(T,"OK")) are serial object well-formed. Also, since perform(ξ) is a behavior of S_X, so are β and β', since a deposit can always occur. Finally, the balance left after each of β and β' is $(x+b+b'), where $x is the balance after perform(ξ), so β and β' are equieffective.

Also, if T and T' are distinct accesses of the kind withdraw_$a and withdraw_$b respectively, then we claim that (T,"OK") and (T',"FAIL") commute. The reason is that if perform(ξ(T,"OK")) and perform(ξ(T',"FAIL")) are both serial object well-formed behaviors then we must have $a \leq x < b$, where $x is the balance after perform(ξ). Then both perform(ξ(T,"OK")(T',"FAIL")) and perform(ξ(T',"FAIL")(T,"OK")) are serial object well-formed behaviors of S_X that result in a balance of $(x - a), and so are equieffective.

On the other hand, if T and T' are distinct accesses of the kind withdraw_$a and withdraw_$b respectively, then (T,"OK") and (T',"OK") do not commute, since if perform(ξ) leaves a balance of $x, where $\max(a,b) \leq x < a+b$, then perform(ξ(T,"OK")) and perform(ξ(T',"OK")) can be serial object well-formed behaviors of S_X, but the sequence

[8] This definition is more complicated than that often used in the classical theory, because we deal with types whose accesses may be specified to be partial and nondeterministic, that is, the return value may be undefined or multiply-defined from a given state.

perform(ξ(T,"OK")(T',"OK")) is not a behavior, since after perform(ξ(T,"OK")) the balance left is $(x - a), which is not sufficient to cover the withdrawal of $b.

7 GENERAL COMMUTATIVITY-BASED LOCKING

In this section, we present our general commutativity-based locking algorithm. The algorithm is described as a generic system. The system type and the transaction automata are assumed to be fixed, and are the same as those of the given serial system. The generic controller automaton has already been defined. Thus, all that remains is to define the generic objects. We define the appropriate objects here, and show that they are dynamic atomic.

7.1 Locking Objects

For each object name X, we describe a generic object automaton L_X (a "locking object"). The object automaton uses the commutativity relation between operations to decide when to allow operations to be performed.

Automaton L_X has the usual signature of a generic object automaton for X. A state s of L_X has components s.created, s.commit-requested and s.intentions. Of these, created and commit-requested are sets of transactions, initially empty, and intentions is a function from transactions to sequences of operations of X, initially mapping every transaction to the empty sequence λ. When (T,v) is a member of s.intentions(U), we say that U *holds a* (T,v)-*lock*. Given a state s and a transaction name T we also define the sequence *total*(s,T) of operations by the recursive definition total(s,T_0) = s.intentions(T_0), total(s,T) = total(s,parent(T))s.intentions(T). Thus, total(s,T) is the sequence of operations obtained by concatenating the values of intentions along the chain from T_0 to T, in order.

The transition relation of L_X is given by all triples (s',π,s) satisfying the following preconditions and effects, given separately for each π. As a convention, any component of s not mentioned in the effect is the same in s as in s'.

CREATE(T), T an access to X
Effect:
s.created = s'.created $\cup$ T

INFORM_COMMIT_AT(X)OF(T), T $\neq T_0$
Effect:
s.intentions(T) = λ
s.intentions(parent(T)) = s'.intentions(parent(T))s'.intentions(T)

s.intentions(U) = s'.intentions(U) for U $\neq$ T, parent(T)

INFORM_ABORT_AT(X)OF(T), T $\neq T_0$
Effect:
s.intentions(U) = λ, U $\in$ descendants(T)
s.intentions(U) = s'.intentions(U), U $\notin$ descendants(T)

REQUEST_COMMIT(T,v), T an access to X
Precondition:
T $\in$ s'.created $-$ s'.commit-requested
(T,v) commutes with every (T',v') in s'.intentions(U), where U $\notin$ ancestors(T)
perform(total(s',T)(T,v)) $\in$ finbehs(S_X)
Effect:
s.commit-requested = s'.commit-requested $\cup$ T
s.intentions(T) = s'.intentions(T)(T,v)
s.intentions(U) = s'.intentions(U) for U $\neq$ T

Thus, when an access transaction is created, it is simply added to the set created. When L_X is informed of a commit, it passes any locks held by the transaction to the parent, appending them at the end of the parent's intentions list. When L_X is informed of an abort, it discards all locks held by descendants of the transaction. A response containing return value v to an access T can be returned only if the access has been created but not yet responded to, every holder of a "conflicting" (that is, non-commuting) lock is an ancestor of T, and perform(T,v) can occur in a move of S_X from a state following the behavior perform(total(s',T)). When this response is given, T is added to commit-requested and the operation (T,v) is appended to intentions(T) to indicate that the (T,v)-lock was granted. It is easy to see that L_X is a generic object i.e, that L_X has the correct external signature and preserves generic object well-formedness.

In [3] we prove the following result.

Theorem 5 *L_X is dynamic atomic.*

An immediate consequence of Theorem 5 and the Dynamic Atomicity Theorem is that if $\mathcal{S}$ is a generic system in which each generic object is a locking object, then $\mathcal{S}$ is serially correct for all non-orphan transaction names.

7.2 Implementations

The locking object L_X is quite nondeterministic; implementations[9] of L_X can be designed that restrict the nondeterminism in various ways, and correctness of such algorithms follows immediately from the correctness of L_X, once the implementation relationship has been proved.[10]

As a trivial example, consider an algorithm expressed by a generic object that is just like L_X except that extra preconditions are placed on the REQUEST_COMMIT(T,v) action, say requiring that no lock at all is held by any non-ancestor of T. Every behavior of this generic object is necessarily a behavior of L_X, although the converse need not be true. That is, this object implements L_X and so is dynamic atomic.

For another example, note that our algorithm models both choosing a return value, and testing that no conflicting locks are held by non-ancestors of the access in question, as preconditions on the single REQUEST_COMMIT event for the access. Traditional database management systems have used an architecture in which a lock manager first determines whether an access is to proceed or be delayed, and only later is the response determined. In such an architecture, it is infeasible to use the return value in determining which activities conflict. We can model such an algorithm by an automaton in which the granting of locks by the lock manager is an internal event whose precondition tests for conflicting locks using a "conflict table" in which a lock for access T is recorded as conflicting with a lock for access T' whenever there are any return values v and v' such that (T,v) does not commute with (T',v'). Then we would have a REQUEST_COMMIT action whose preconditions include that the return value is appropriate and that a lock had previously been granted for the access. If we do this, we obtain an object that can be shown to be an implementation of L_X, and therefore its correctness follows from that of L_X.

Many slight variations on these algorithms can be considered, in which locks are obtained at different times, recorded in different ways, and tested for conflicts using different relations; so long as the resulting algorithm treats non-commuting operations as conflicting, it should not be hard to prove that these algorithms implement L_X, and so are correct. Such implementations could exhibit much less concurrency than L_X, because they use a coarser test for deciding when an access may proceed. In many cases the loss of potential concurrency might be justified by the simpler computations needed in each indivisible step.

Another aspect of our algorithm that one might wish to change in an implementation is the complicated data structure maintaining the "intentions", and the corresponding need to replay all the operations recorded there when determining the response to an access. In [3] we consider Moss' algorithm, which is able to summarize all these lists of operations in a stack of versions of the serial object, at the cost of reducing available concurrency by

[9] Recall that "implementation" has a formal definition. The implementation relation only relates external behaviors, but allows complete freedom in the choice of automaton states.

[10] In [3] we give some techniques that can be used to prove an implementation relationship between two automata.

using a conflict relation in which all updates exclude one another.

References

[1] J. Aspnes, A. Fekete, N. Lynch, M. Merritt, and W. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Proceedings of 14th International Conference on Very Large Data Bases*, pages 431–444, August 1988.

[2] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Nested transactions and read/write locking. In *6th ACM Symposium on Principles of Database Systems*, pages 97–111, San Diego, CA, March 1987. Expanded version available as Technical Memo MIT/LCS/TM-324, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, April 1987.

[3] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. Technical Memo MIT/LCS/TM-370, Massachusetts Institute Technology, Laboratory for Computer Science, August 1988. A revised version will appear in JCSS.

[4] K. Goldman and N. Lynch. Nested transactions and quorum consensus. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*, pages 27–41, August 1987. Expanded version is available as Technical Report MIT/LCS/TM-390, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, May 1987.

[5] M. Herlihy, N. Lynch, M. Merritt, and W. Weihl. On the correctness of orphan elimination algorithms. In *Proceedings of 17th IEEE Symposium on Fault-Tolerant Computing*, pages 8–13, 1987. Also, MIT/LCS/TM-329, MIT Laboratory for Computer Science, Cambridge, MA, May 1987. To appear in Journal of the ACM.

[6] B. Liskov. Distributed computing in argus. *Communications of ACM*, 31(3):300–312, March 1988.

[7] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. A theory of atomic transactions. In *International Conference on Database Theory*, pages 41–71, Bruges, Belgium, September 1988. LNCS 326, Springer Verlag.

[8] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA., April 1987.

[9] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute Technology, 1981. Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute Technology, April 1981. Also, published by MIT Press, March 1985.

[10] A. Spector and K. Swedlow. Guide to the camelot distributed transaction facility: Release 1, October 1987. Available from Carnegie Mellon University, Pittsburgh, PA.

[11] W.E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, Massachusetts Institute Technology, 1984. Technical Report MIT/LCS/TR-314, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, March 1984.