LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-421

A SERIALIZATION GRAPH CONSTRUCTION FOR NESTED TRANSACTIONS

Alan Fekete
Nancy A. Lynch
William E. Weihl

February 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Serialization Graph Construction for Nested Transactions

Alan Fekete*
University of Sydney
Sydney, Australia

Nancy Lynch†
William E. Weihl‡
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

This paper makes three contributions. First, we present a proof technique that offers system designers the same ease of reasoning about nested transaction systems as is given by the classical theory for systems without nesting, and yet can be used to verify that a system satisfies the robust "user view" definition of correctness of [10]. Second, as applications of the technique, we verify the correctness of Moss' read/write locking algorithm for nested transactions, and of an undo logging algorithm that has not previously been presented or proved for nested transaction systems. Third, we make explicit the assumptions used for this proof technique, assumptions that are usually made *implicitly* in the classical theory, and therefore we clarify the type of system for which the classical theory itself can reliably be used.

Keywords: concurrency control, recovery, fault-tolerance, nested transactions, serializability, verification.

1 Introduction

The 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 of organizing distributed systems in which information is maintained in persistent modifiable objects. Nested transactions allow the benefits of atomicity to be used within a transaction, so that, for example, a transaction can include several simultaneous remote procedure calls, which can be coded without considering possible interference among them. Examples of systems using nested transactions are Argus [9] and Camelot [15]. In

*Supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

†Supported in part by the National Science Foundation under Grant CCR-86-11442, in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and in part by the Office of Naval Research under Contract N00014-85-0168.

‡Supported in part by the National Science Foundation under Grant CCR-8716884, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988.

these systems "atomic" objects can be created and operations on these objects are guaranteed to be serializable, even though they execute concurrently. In both Argus and Camelot the default algorithm used for concurrency control and recovery is the locking protocol of Moss [13], but the implementor of an object has the option of writing his or her own concurrency control and recovery routines.

A natural question is what "correctness" means for concurrency control and recovery algorithms in a nested transaction system. Once a specification for correct functioning has been given, one seeks to prove that existing algorithms are correct. Moreover, the possibility of user-defined concurrency control in a system leads one to seek proof methods that are modular, so that when one object is reimplemented (for performance reasons) in a previously correct system, the new system may be proved correct without needing to reconsider those parts that have not changed. Such specification and proof issues have been addressed in a major research project started by Lynch and Merritt [10] and continued in [5, 6, 8, 4, 1, 11].

In [10], a notion of correctness called "serial correctness" is defined for nested transaction systems. The definition of "serial correctness" presented there is a "user view" specification: the users of the transaction system should only be able to observe behavior that they could observe when interacting with a system in which their transactions were run without concurrency and without failure after partial activity. The definition of serial correctness embodies not only the serializability condition of the classical theory, but also the "external consistency" condition, i.e., that the apparent serial execution must not reverse the order of any pair of transactions for which one completed before the other was invoked. Also, unlike the classical theory, this definition of serial correctness is explicitly formulated to apply to systems in which transactions can abort; in the classical theory, aborts are handled by considering only executions in which all transactions commit.[1]

The definition of serial correctness from [10] is used in [10, 5], where a proof technique is developed for verifying Moss' algorithm. Modular proof techniques for locking algorithms can be found in [4]. The same definition and proof techniques have been used in proofs of the correctness of several other kinds of transaction-processing algorithms, including multiversion timestamp-based algorithms for concurrency control and recovery [1], algorithms for management of replicated data [6], and algorithms for management of orphan transactions [8]. The proof techniques of these papers are very general. They apply to large classes of systems, including those where different data objects are implemented independently, and where the type of the objects can be used to obtain increased concurrency (as in [17]). We summarize the system model, definition of serial correctness, and main proof technique in the next section.

We can contrast the development of our theory of serial correctness for nested transaction systems with the classical serializability theory for systems without transaction nesting, as presented (for example) in [14] and [3]. The classical theory uses a system model and correctness definition that are somewhat more restrictive than necessary; for example, the classical correctness definition is not stated in terms of the user view of the system, but rather in terms of the activity at the data objects. The classical model and definition work very well for a number of simple update-in-place algorithms, but a different *definition* of correctness is needed to cope with multiversion algorithms, and yet another for replication management. The classical theory is also restricted in that it deals almost exclusively with data objects allowing only read and write operations.

[1] As discussed in [16], aborts must be modeled explicitly to analyze the subtle interactions between concurrency control and recovery. Because it does not model aborts explicitly, and implicitly assumes an "update-in-place" model for recovery, the classical theory is not general enough to model certain kinds of algorithms.

An advantage of the classical theory, however, is that for the simplest concurrency control algorithms such as two-phase locking or single-version timestamps, it yields extremely simple and intuition-supporting proofs. These proofs are based on the absence of cycles in a "serialization graph," a graph whose nodes are the transactions and whose edges record conflicts between activity of the transactions.

We would like to be able to combine the best features of both theories. In particular, we would like to be able to use serialization graph proof techniques similar to those of the classical theory to reason about nested transaction systems, wherever this is possible. We would especially like to use such techniques to prove that such systems satisfy the user view serial correctness condition of [10]. We would also like to extend the applicability of serialization graph techniques to data objects that admit other kinds of operations besides reads and writes. In this paper, we show how to combine the two theories in these ways.

More specifically, we develop a proof technique for nested transaction systems in which proofs have the same simple form as in the classical theory, namely, one must show that a graph (having transactions for nodes, and edges representing necessary ordering between transactions) is acyclic. Thus, we define a new kind of "serialization graph" and prove that, under certain assumptions, the absence of cycles in this graph is a sufficient condition to ensure the serial correctness of a system. In the first part of the paper, we restrict our attention to systems in which each data object admits only read and write operations. For such systems, we assume that (once aborted transactions' activity is ignored) a read operation always returns the value written by the most recent write operation. This assumption is true of systems in which each data object is stored in a single location that is overwritten by any write access, and where an underlying recovery system restores the appropriate old value when an ancestor of the most recent write is aborted.

In much of the classical work on database concurrency control, these restrictions and assumptions are made early on, and in fact the definition of correctness often includes them. Systems satisfying these assumptions are very common, and while we feel that it is inappropriate to make these assumptions when defining the correctness condition to be satisfied, it is clearly useful to find a simple sufficient condition that guarantees correctness when the system does satisfy them.

We note that in contrast to the classical theory, the acyclicity of the graphs we construct is merely a sufficient condition for serial correctness, rather than necessary and sufficient. This is primarily because our notion of serial correctness, based as it is on the user's view of the system, is not as restrictive as the one used in the classical theory.

After presenting our results for reads and writes, we indicate how they can be generalized to arbitrary data types. That is, we define serialization graphs for systems with objects of arbitrary data type, and prove once again that absence of cycles implies serializability. Once again, the values returned by accesses to objects are assumed to satisfy special restrictions.

We use our serialization graphs to prove correctness of two algorithms—the read/write locking algorithm of Moss and an undo logging algorithm. (The latter algorithm is a generalization to nested transaction systems of an algorithm due to Weihl [16]).

Other work has also been done on modeling nested transaction systems. Hadzilacos and Hadzilacos [7] present a generalization of the classical theory to handle "object bases," which exhibit a nesting structure very much like that considered in this paper. (Our objects correspond to the instance variables in their objects, and our accesses to objects correspond to the local steps that access the instance variables.) They define a serialization graph construction,

and give an acyclicity condition for serializability. However, they do not consider recovery[2], and their basic model is significantly less general than ours (for example, their correctness condition is appropriate only with an update-in-place single-version implementation of objects, while we permit multi-version implementations). Beeri, Bernstein and Goodman [2] present proof techniques that are useful for systems organized using multiple levels of abstraction, with concurrency control performed separately at each level. The nesting in such systems corresponds to levels of data abstraction, while the nesting considered here corresponds more to levels of procedural abstraction. It may be that the techniques in [2] could be applied to the kinds of systems we consider here, but their techniques are more complicated, allowing replacement of entire subtrees of nested activity by single actions as well as the reordering of actions in a history. Also, they do not present a simple acyclic graph condition for correctness, and they do not model recovery in their work.

The remainder of this paper is organized as follows. First, in Section 2, we summarize our earlier work on which this work is based. Then, in Section 3, we give the assumptions we make for systems based on read/write objects; that is, we define such systems and define the condition that says that all reads return the latest value. In Section 4, we present our serialization graph construction and the theorem that says that acyclicity of the serialization graph implies serial correctness. In Section 5, we give a proof of Moss' algorithm using our serialization graphs. In Section 6, we indicate how to extend the work to other data types besides read/write objects; this section includes a description and proof of the general undo logging algorithm. Finally, we conclude with a discussion and some suggestions for further work.

2 Background

In this section, we summarize the main concepts from our earlier work that are used in the rest of the paper. Complete details can be found in [11]. The reader who is already familiar with our work, or who is not interested in the details of the proofs, may skip or skim this section.

2.1 Review: 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 [12] and [11].

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 actions*. We refer to both input and output actions as *external actions*. We use the terms $in(A)$, $out(A)$, and $ext(A)$ 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.

[2] A later manuscript of their paper has extended the results to include recovery.

[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.

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 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.

2.2 Review: 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 [11].

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 op-

eration 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*, *ancestor*, *lca* (that is, least common 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 (henceforth simply called *values*). If T is a transaction name that is an access to the object name X and v is a 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.

2.2.1 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 value
Output:

REQUEST_CREATE(T'), for T' a child of T
REQUEST_COMMIT(T,v), for v a 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. 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 systems 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 [11].

2.2.2 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 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)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.[4]

[4]This is formally defined in [11] and means that the object does not violate well-formedness unless its environment has done so first.

2.2.3 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 [11].

2.2.4 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[5] 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 $transaction(\pi) = T$, and we define $\beta|X$ to be the subsequence of β consisting of those serial actions π such that $object(\pi) = X$. We define $serial(\beta)$ 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. We say the T is *live* in β if β contains a CREATE(T) event but does not contain a completion event for T.

2.2.5 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.

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.

2.3 Review: 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 [11], and is an extension to nested transaction systems of ideas presented in [18, 17]. 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 2. 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.

2.3.1 Simple Systems

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.

[5] 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 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 CREATE, ABORT or COMMIT events 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 [11]. 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.

2.3.2 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 [11].

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,[6] 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 2. 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).

The following lemma will be used later in proving that certain sibling orders are suitable:

Lemma 1 *Let β be a sequence of serial events and let A be an irreflexive partial order on the events in β. Let R be a sibling order satisfying the following condition: If π and π' are events in β such that $(\pi, \pi') \in A$ and lowtransaction(π) is neither an ancestor nor a descendant of lowtransaction(π'), then $(\pi, \pi') \in R_{event}(\beta)$. Then $R_{event}(\beta)$ and A are consistent partial orders on the events of β.*

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 Serializability 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

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

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 $view(\beta, T, R, X)$ to be $perform(\xi)$.

Theorem 2 *(Serializability Theorem[11])*
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, $view(\beta, T, R, X) \in finbehs(S_X)$. Then β is serially correct for T.

3 Assumptions

In this section, we present our two main assumptions. First, for all of this paper except Section 6, we will assume that the fixed serial system (with respect to which serial correctness is defined) contains only objects of a particularly simple type, where the only ways to access an object are to read it or to write it. This assumption reflects the reality at the lowest level of many database management systems, since these are the only accesses possible to a disk. While many systems do contain more complicated data types at a higher level of abstraction (for example, in a relational database the accesses at the conceptual level include joins, selections, etc.) the assumption that all the objects have this simple type is usually made in the classical development of serializability theory, and we make it here to show the relationships between our results and the classical theory. In Section 6 we remove this assumption.

3.1 Read/Write Serial Objects

Formally, our first assumption is that every serial object in the serial system is a specific kind of object, described below, which we call a "read/write object." That is, for each object name X there is a domain of values $\mathcal{D}$, a function *kind* (which indicates for each access whether it is a read or a write), a function *data* (which indicates for each write access the value written—in our model, all parameters of an access are regarded as encoded in its name, so this function serves to decode one parameter), and an initial value d, such that the serial object automaton S_X has the following state and transition relation. Its state contains two components: *active* (either *nil*, or the name of an access to X) and *data* (an element of $\mathcal{D}$, representing the most recently written value). The start state s has $s.active = nil$, and $s.data = d$. The transition relation is as follows:

CREATE(T), T an access to X
  Effect:
    $s.active = T$

REQUEST_COMMIT(T,v), T a write access to X
  Precondition:
    $s'.active = T$
    $v = OK$
  Effect:
    $s.active = nil$
    $s.data = data(T)$

REQUEST_COMMIT(T,v), T a read access to X
  Precondition:
    $s'.active = T$
    $s'.data = v$
  Effect:
    $s.active = nil$

The definition of the automaton S_X ensures that, in a serial system, each read access returns the most recent value written. This can be seen from the effects of a REQUEST_COMMIT for a write access, which stores the value written by the access in the state component *data*, and from the preconditions for a REQUEST_COMMIT for a read access, which ensure that the value returned is the value of the state component *data*.

In the sequel, we will need a definition for the "final value" of a read/write object after a sequence of write accesses. If β is a sequence of serial actions and X is an object name, we define *write-sequence*(β, X) to be the subsequence of β consisting of REQUEST_COMMIT events for transactions that are write accesses to X; then we define *last-write*(β, X) to be *transaction*(π) where π is the last event in *write-sequence*(β, X) (if *write-sequence*(β, X) is empty, *last-write*(β, X) is undefined.) Finally, we define *final-value*(β, X) to be the initial value d if *last-write*(β, X) is undefined, and *data*(*last-write*(β, X)) otherwise. Thus, *final-value*(β, X) is the latest value written in β for X. The following lemmas characterize the state and behaviors of the read/write object S_X in terms of *final-value*:

Lemma 3 *Let β be a finite schedule of read/write serial object S_X, and let s be the (unique) state of S_X after β. Then $s.data$ = final-value(β, S_X).*

Lemma 4 *Let β be a finite behavior of S_X. Then βperform(T, v) is a behavior of S_X exactly when either T is a write access to X and $v = OK$, or T is a read access to X and $v =$ final-value(β, X).*

3.2 Appropriate Return Values

In a real transaction-processing system, different transactions can access an object concurrently. Concurrency control and recovery algorithms are needed to ensure that the effect of a concurrent execution is the same as that of some execution of the serial system, as far as the users of the system can observe. Rather than developing a complex model of a real transaction-processing system, we prove results about behaviors of simple systems satisfying certain restrictions; we then show that a particular real transaction-processing system implements the simple system (so each of its behaviors is also a simple behavior) and that its behaviors satisfy the necessary restrictions. One advantage of this approach is that it allows us to make very few assumptions about the *structure* of a transaction-processing system; instead, we make assumptions about its behaviors, represented as simple behaviors.

In defining these assumptions, and in the remainder of the paper, we will apply the definitions above of *write-sequence*, *last-write*, and *final-value* to behaviors of simple systems. Notice that each of these was defined in terms of general sequences of serial actions, so applying them to simple behaviors does not cause any problems.

Our first assumption described above, namely that each serial object is a read/write object, applies to serial systems. Our second assumption applies to behaviors of simple systems. Informally, we assume the existence of some underlying recovery system that ensures that descendants of aborted and uncommitted transactions appear never to have happened; once the

actions of these transactions have been removed from consideration, the return value for an access is what one would expect from a simplistic model of the simple system, where each object's value is stored in a location, being overwritten with a new value by write accesses and unaffected by read accesses. Much of the classical work on concurrency control has used this simplistic model without comment.

To make this formal, we introduce a definition: if β is a simple behavior, then we say that β *has appropriate return values* provided that whenever π is a REQUEST_COMMIT(T,v) event occurring in $visible(\beta, T_0)$ and T is an access to an object X, then either T is a write access and $v = OK$, or T is a read access and $v = final\text{-}value(\delta, X)$, where δ is the prefix of $visible(\beta, T_0)$ preceding π. Notice that we here restrict attention to the part of the sequence β that is visible to T_0. This restriction corresponds to the classical theory's focus on the "permanent" part of the computation (called the "committed projection" in [3])—the part that has committed to the outside world.

The following is a convenient characterization of appropriate return values for systems in which all serial objects are read/write objects.

Lemma 5 *Let β be a finite simple behavior. Then β has appropriate return values if and only if $perform(operations(visible(\beta, T_0))|X)$[7] is a behavior of S_X for all X.*

Proof: Suppose β has appropriate return values and X is an object name. We must show that $perform(operations(visible(\beta, T_0)|X))$ is a behavior of S_X. We show the equivalent statement that for any prefix ξ of $operations(visible(\beta, T_0)|X)$, $perform(\xi)$ is a behavior of S_X, which we do by induction on the number of operations in ξ. The base case, when there are no operations, is trivial. Otherwise $\xi = \xi'(T, v)$. By the induction hypothesis, $perform(\xi')$ is a behavior of S_X. Now since (T, v) is in $operations(visible(\beta, T_0)|X)$, we see that T is an access to X and there is an event π = REQUEST_COMMIT(T,v) in $visible(\beta, T_0)$. Since β has appropriate return values, either T is a write access and $v = OK$, or T is a read access and $v = final\text{-}value(\delta, S_X)$, where δ is the prefix of $visible(\beta, T_0)$ preceding π. In the case where T is a read access, then we note that $write\text{-}sequence(\delta, S_X) = write\text{-}sequence(perform(\xi'), S_X)$ and so $final\text{-}value(\delta, S_X) = final\text{-}value(perform(\xi'), S_X)$. Thus $perform(\xi(T, v))$ is a behavior of S_X by Lemma 4.

Conversely, suppose $perform(operations(visible(\beta, T_0)|X))$ is a behavior of S_X for all X. Consider π, a REQUEST_COMMIT(T,v) event occurring in $visible(\beta, T_0)|X$ where T is an access. We must have (T, v) in $operations(visible(\beta, T_0)|X)$, where $object(T) = X$. Let ξ' be the prefix of $operations(visible(\beta, T_0)|X)$ preceding (T, v). Since $perform(\xi'(T, v))$ is a behavior of S_X, by Lemma 4 we conclude that either T is a write access and $v = OK$, or T is a read access and $v = final\text{-}value(perform(\xi'), S_X)$. However, we note that if δ is the prefix of $visible(\beta, T_0)$ preceding π, then $write\text{-}sequence(\delta, S_X) = write\text{-}sequence(perform(\xi'), S_X)$ and so $final\text{-}value(\delta, S_X) = final\text{-}value(perform(\xi'), S_X)$. Thus, either T is a write access and $v = OK$, or T is a read access and $v = final\text{-}value(\delta, S_X)$. Since π was arbitrary, β has appropriate return values. □

3.3 A Sufficient Condition for Appropriate Return Values

The hypothesis that a system's behaviors have appropriate return values is commonly made, and in the classical development of serializability theory it is usually regarded as axiomatic.

[7] An "operation" is a pair (T, v); the operator "operations" extracts the sequence of operations corresponding to the REQUEST_COMMIT events in an event sequence.

However when one studies or designs a real system one must consider how particular algorithms lead to this hypothesis being met. For write accesses it is certainly easy to ensure that the return value is OK. However the situation with read accesses is very different. In this section, we define simple conditions that are sufficient to ensure appropriate return values. While these conditions are not only sufficient and not necessary, they do apply to many algorithms.

We need to show the following: for every REQUEST_COMMIT(T,v) event π in *visible*(β, T_0), where T is a read access to X, the return value v is equal to *final-value*(δ, X), where δ is the prefix of visible(β,T_0) preceding π. Now, at the time π occurs, the sequence δ is not yet determined, since it depends on all the COMMIT events in β, including those that follow π. It is useful to have conditions that can be checked when π occurs and that are sufficient to ensure appropriate return values. We define two conditions. The first requires that the return value for a REQUEST_COMMIT event be "current" using the sequence of events that occur before the REQUEST_COMMIT event. Informally, a REQUEST_COMMIT event for a read access is current if the return value provides the appearance of accessing a variable that is overwritten when each new write access requests to commit and is restored when a transaction ABORT occurs in order to remove all trace of the descendants of the aborted transaction. The second condition requires that the return value be "safe," in the sense that all the needed COMMIT events are already present in the sequence before the REQUEST_COMMIT. Informally, a REQUEST_COMMIT event for a read access is safe if the writer of the current value (under the assumption that there is a current value that is overwritten and restored) is visible to the reader. This ensures that any ancestor of the writer that is not yet committed is also an ancestor of the reader. Thus, the writer cannot be aborted (by aborting one of its ancestors) without also aborting the reader. A read access that is not safe is sometimes described as reading "dirty data."

More formally, if β is any sequence of serial actions, we define *clean*(β) to be the subsequence of β containing all events whose *hightransactions* are not orphans in β. Then if β is a sequence of serial actions and X is an object name, we define *clean-write-sequence*(β, X) to be *write-sequence*(*clean*(β), X). Also, we define *clean-last-write*(β, X) to be *last-write*(*clean*(β), X). Similarly, we define *clean-final-value*(β, X) to be *final-value*(*clean*(β), X).

Now, if β is a sequence of serial actions and π is a REQUEST_COMMIT(T,v) event that appears in β, where T is a read access to X, then we say that π is *current* in β if v = *clean-final-value*(β', X), where β' is the longest prefix of β that does not include the event π. In addition, if β is a sequence of serial actions and π is a REQUEST_COMMIT(T,v) event that appears in β, where T is a read access to X, then we say that π is *safe* in β if *clean-last-write*(β', X) is either undefined or visible to T in β', where β' is the longest prefix of β that does not include the event π.

We have the following key lemma.

Lemma 6 *Let β be a simple behavior such that the following hold.*

1. *If π is a REQUEST_COMMIT(T,v) event that occurs in visible(β, T_0) where T is a write access to X, then $v = OK$.*
2. *If π is a REQUEST_COMMIT(T,v) event that occurs in visible(β, T_0) where T is a read access to X, then π is current and safe in β.*

Then β has appropriate return values.

Proof: Condition (1) above is the first condition needed to argue that β has appropriate return values. It remains to show that if π a REQUEST_COMMIT(T,v) event that occurs

in *visible*(β, T_0) where T is a read access to X, and π is current in β and safe in β, then $v =$ *final-value*(δ, S_X) where δ is the prefix of *visible*(β, T_0) preceding π.

Now, if π is current in β then by definition $v =$ *clean-final-value*(β', S_X) where β' is the prefix of β preceding π. Thus we need only prove that *clean-last-write*$(\beta', S_X) =$ *last-write*(δ, S_X). Since β is a simple behavior (and so does not contain both a COMMIT and an ABORT for any transaction), any transaction that is visible to T_0 in β is not an orphan in β, and hence is not an orphan in β'. Thus *write-sequence*(δ, S_X) is a subsequence of *clean-write-sequence*(β', S_X).

We will show that the last event in *clean-write-sequence*(β', S_X) if any, does occur in δ. Note that this last event is a REQUEST_COMMIT for *clean-last-write*(β', S_X). By the hypothesis that π is safe, we see that *clean-last-write*(β', S_X) is visible to T in β', and hence in β. Since π occurs in *visible*(β, T_0) we have that T is visible to T_0 in β. We deduce that *clean-last-write*(β', S_X) is visible to T_0 in β, and so the last event in *clean-write-sequence*(β', S_X) occurs in *visible*(β, S_X). Since it precedes π, it occurs in δ as claimed. Now (as it is a REQUEST_COMMIT for a write access to X) we can deduce it will occur in *write-sequence*(δ, S_X). Further, since the order of events in *write-sequence*(δ, S_X) is the same as the order of those events in *clean-write-sequence*(β', S_X) (each order is just the order in β), it must be the last event as required. □

4 The Serialization Graph Construction

In this section, we present our serialization graph construction. Recall that the serial correctness condition of [11] embodies not only the serializability condition of the classical theory, but also the external consistency condition. Therefore, our serialization graphs will have two kinds of edges, "conflict edges" and "precedence edges." The former are similar to those used in the classical theory, and serve to fix the order of conflicting operations. The latter are added to capture restrictions required for external consistency.

We define a *conflict* relation between accesses so that two write accesses to the same object conflict, as do a write and a read access to the same object, but not two read accesses or two accesses to different objects. More formally, let S_X be a serial object for object name X, and let T and T' are accesses to X. Then we say that T and T' *conflict* if either T or T' is a write access.

We extend the preceding definition to a conflict relation on *operations*: if S_X is a serial object for object name X, (T, v) and (T', v') are operations where T and T' are accesses to X, then we say that (T, v) and (T', v') *conflict* if and only if T and T' conflict. The following proposition shows that non-conflicting operations can be reordered in serial behaviors:

Proposition 7 *Suppose that ξ is a sequence of operations of X such that perform(ξ) is a serial object well-formed behavior of S_X. Suppose that η is a reordering of ξ such that all pairs of conflicting operations occur in the same order in η and in ξ. Then perform(η) is a behavior of S_X.*

We next derive a conflict relation between sibling transactions, based on conflicts between descendant operations. Formally, if β is a sequence of serial actions, we define *conflict*(β) to be the relation such that $(T, T') \in$ *conflict*(β) if and only if T and T' are siblings and the following holds: there are events ϕ and ϕ' in *visible*(β, T_0) such that $\phi =$ REQUEST_COMMIT(U,v) where U is a descendant of T, $\phi' =$ REQUEST_COMMIT(U',v') where U' is a descendant of T', (U, v) conflicts with (U', v') and ϕ precedes ϕ' in *visible*(β, T_0). Informally, T conflicts with T' if a descendant of T' accesses some object X after a descendant of T accesses X in a

conflicting manner (i.e., at least one access is a write). Note that if two siblings are related by *conflict*(β) then they (and thus their common parent) are visible to T_0 in β.

If β is a sequence of serial actions, define *precedes*(β) to be the relation such that $(T, T') \in$ *precedes*(β) if and only if T and T' are siblings whose common parent is visible to T_0 in β, and a report event for T and a REQUEST_CREATE(T') occur in β, in that order. Informally, T precedes T' if their parent knows that T finished before it requests the creation of T'.

If β is a sequence of serial actions, we incorporate the information in the relations *conflict*(β) and *precedes*(β) into a graph, as follows. We define the serialization graph $SG(\beta)$ to be the union of a collection of disjoint directed graphs $SG(\beta, T)$, one for each transaction T that is visible to T_0 in β. The graph $SG(\beta, T)$ has nodes labelled by the children of T, and a directed edge from the node labelled T' to the node labelled T'' if and only if T' and T'' are children of T and $(T', T'') \in$ *precedes*(β) $\cup$ *conflict*(β).

The following theorem gives a sufficient condition for a sequence β of serial actions to be serially correct for T_0. It relies on our Serializability Theorem (Theorem 2).

Theorem 8 *Let β be a finite simple behavior that has appropriate return values. Suppose that $SG(\beta)$ is acyclic. Then β is serially correct for T_0.*

Proof: For each transaction T that is visible to T_0 in β, we can choose some total order on the children of T that is a topological sort of the directed graph $SG(\beta, T)$, since that graph is acyclic. Let R denote the sibling order given by the union of the chosen total orders. We claim that R is suitable for β (as defined in Section 2.3) and that for every object name X, *view*(β, T_0, R, X) is a behavior of S_X. Once we have shown the truth of these claims, Theorem 2 (the Serializability Theorem of [11]—in Section 2.3) completes the proof.

To show that R is suitable we need to check that it orders all pairs of siblings T and T' that are lowtransactions of events in *visible*(β, T_0), and that $R_{event}(\beta)$ and *affects*(β) are consistent partial orders on the events in *visible*(β, T_0).

By construction, R orders all pairs of siblings whose common parent is visible to T_0 in β. We argue that this includes all pairs of siblings that are lowtransactions of actions in *visible*(β, T_0) as follows: the hightransaction of an action in *visible*(β, T_0) is visible to T_0 in β, and the parent of an action's lowtransaction is either the action's hightransaction (for completion actions) or the parent of the action's hightransaction (for other actions). Since the action's hightransaction is visible to T_0 in β, so is the parent of the action's hightransaction. Thus R orders all pairs of siblings T and T' that are lowtransactions of events in *visible*(β, T_0).

Suppose that π and π' are events in *visible*(β, T_0) such that π affects π' in β and *lowtransaction*(π) is neither an ancestor nor a descendant of *lowtransaction*(π'). It is easy to show that there must be a common ancestor T of *lowtransaction*(π) and *lowtransaction*(π') such that a report event for T_1 precedes a REQUEST_CREATE(T_2) event in β, where T_1 and T_2 are the children of T that are ancestors of *lowtransaction*(π) and *lowtransaction*(π'), respectively. It follows that $(T_1, T_2) \in$ *precedes*(β). Since R was chosen using a topological sort of the graphs $SG(\beta, T)$, *precedes*(β) $\subseteq R$. Thus $(T_1, T_2) \in R$, and so $(\pi, \pi') \in R_{event}(\beta)$. It follows from Lemma 1 that $R_{event}(\beta)$ and *affects*(β) are consistent partial orders on the events in *visible*(β, T_0). Thus R is suitable for β.

Now let X be an object name. We must show that $\gamma =$ *view*(β, T_0, R, X) is a behavior of S_X. Lemma 5 implies that *perform*(*operations*(*visible*(β, T_0)$|X$)) is a behavior of S_X. Now γ is of the form *perform*($(T_1, v_1)(T_2, v_2) \ldots (T_n, v_n)$), where the (T_i, v_i) are the operations of X that occur in *visible*(β, T_0), and $(T_i, T_{i+1}) \in R_{trans}$ for every i from 1 to $n-1$ inclusive. We make the claim: If T_i conflicts with T_j and $i < j$, then REQUEST_COMMIT(T_i,v_i) precedes

REQUEST_COMMIT(T_j,v_j) in $visible(\beta, T_0)$. In other words, γ can be obtained from $perform(operations(visible(\beta, T_0)|X))$ simply by reordering non-conflicting operations.

The claim is proved as follows: Since REQUEST_COMMIT(T_i, v_i) and REQUEST_COMMIT(T_j, v_j) both occur in $visible(\beta, T_0)$ it is enough to show that REQUEST_COMMIT(T_i, v_i) does not precede REQUEST_COMMIT(T_j, v_j) in $visible(\beta, T_0)$. Suppose it did. Then letting U and U' denote the children of $lca(T_i, T_j)$ that are ancestors of T_i and T_j respectively, we would have $(U', U) \in conflict(\beta)$, and so $(U', U) \in SG(\beta, lca(T_i, T_j))$ and therefore $(U', U) \in R$. Thus $(T_j, T_i) \in R_{trans}$, contradicting $(T_i, T_j) \in R_{trans}$ which follows from $i < j$. Thus the claim is established.

By definition, the operations in $operations(visible(\beta, T_0)|X)$ are exactly the same as those in the sequence $(T_1, v_1)(T_2, v_2) \ldots (T_n, v_n)$. Moreover, as the claim above asserts, conflicting operations occur in the same order. Therefore, by Proposition 7 and the fact that $perform(operations(visible(\beta, T_0)|X))$ is a behavior of S_X, we have that γ is a finite behavior of S_X.

Theorem 2 then implies the result. □

5 Moss' Algorithm

In this section we use the serialization graph described above to prove the correctness of Moss' algorithm for read/write locking [13], the basic concurrency control mechanism in the Argus and Camelot systems.

5.1 Generic Systems

First we describe one way to model a transaction-processing system that includes concurrency control and recovery algorithms. We will model such a system as a "generic system," which is composed of transaction automata, "generic object automata" and a "generic controller." In this paper, we include only a sketch; complete definitions appear in [4].

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 (besides the CREATE and REQUEST_COMMIT actions) special INFORM_COMMIT and INFORM_ABORT input actions to inform it about the completion of (arbitrary) transactions.

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.

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.

5.2 A Read/Write Locking Object Automaton

We model a system using Moss' algorithm as a generic system in which every generic object automaton is the "read/write locking object automaton" $M1_X$ described below, derived from the appropriate serial object S_X. The automaton $M1_X$ maintains a stack of "values," and manages "read locks" and "write locks."[8]

We give here the definition of the read/write locking object $M1_X$. $M1_X$ has the usual external actions for a generic object automaton for X, and it has no internal actions. A state s of $M1_X$ has components *s.created*, *s.commit-requested*, *s.write-lockholders* and *s.read-lockholders*, all sets of transactions, and *s.value*, which is a function from *s.write-lockholders* to $\mathcal{D}$, the domain of basic values. We say that a transaction in *write-lockholders holds a write-lock*, and similarly that a transaction in *read-lockholders holds a read-lock*. The start states of $M1_X$ are those in which *write-lockholders* $= \{T_0\}$ and $value(T_0)$ is d (the initial value of S_X), and the other components are empty.

The transition relation of $M1_X$ is as follows.

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:
if $T \in s'.write\text{-}lockholders$
then
$s.write\text{-}lockholders = (s'.write\text{-}lockholders - \{T\}) \cup \{parent(T)\}$
$s.value(parent(T)) = s'.value(T)$
$s.value(U) = s'.value(U)$,
for $U \in s.write\text{-}lockholders - \{parent(T)\}$
if $T \in s'.read\text{-}lockholders$
then $s.read\text{-}lockholders = (s'.read\text{-}lockholders - \{T\}) \cup \{parent(T)\}$

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Effect:
$s.write\text{-}lockholders = s'.write\text{-}lockholders - descendants(T)$
$s.read\text{-}lockholders = s'.read\text{-}lockholders - descendants(T)$
$s.value(U) = s'.value(U)$ for all $U \in s.write\text{-}lockholders$

REQUEST_COMMIT(T,v), T a read access to X
Precondition:
$T \in s'.created - s'.commit\text{-}requested$
$s'.write\text{-}lockholders \subseteq ancestors(T)$
$v = s'.value(least(s'.write\text{-}lockholders))$
Effect:

[8]This automaton is a simplification of the read/update locking automaton M_X defined in [4]. The latter is defined for any serial object, rather than the particular read/write serial object defined in this paper.

$s.commit\text{-}requested = s'.commit\text{-}requested \cup \{T\}$
$s.read\text{-}lockholders = s'.read\text{-}lockholders \cup \{T\}$

REQUEST_COMMIT(T,v), T a write access to X
Precondition:
$T \in s'.created - s'.commit\text{-}requested$
$s'.write\text{-}lockholders \cup s'.read\text{-}lockholders \subseteq ancestors(T)$
$v = OK$
Effect:
$s.commit\text{-}requested = s'.commit\text{-}requested \cup \{T\}$
$s.write\text{-}lockholders = s'.write\text{-}lockholders \cup \{T\}$
$s.value(T) = data(T)$
$s.value(U) = s'.value(U)$,
for all $U \in s.write\text{-}lockholders - \{T\}$

When an access transaction is created, it is added to the set *created*. When $M1_X$ is informed of a commit, it assigns any locks held by the transaction to the parent, and also assigns any stored value to the parent. When $M1_X$ is informed of an abort, it discards all locks held by descendants of the transaction. A response to an access T can be returned only if the access has been created but not yet responded to, every holder of a conflicting lock is an ancestor of T, and the return value is appropriate, being *OK* for a write access and the value corresponding to the least holder of a write lock if the access is a read. (The component *s.write-lockholders* will always be a linear chain of transactions, with every element being either an ancestor or descendant of every other. The least element of such a set is the unique descendant of all other elements.) When this response is given, T is added to *commit-requested* and granted the appropriate lock. Also, if T is a write access, the new value is stored as $value(T)$, while if T is a read access, no change is made to *value*.

5.3 Basic Properties of $M1_X$.

We begin with some basic properties of $M1_X$. These can be proved by common techniques such as invariant assertions or arguments about sequences of actions.

The statements of the results below use some terminology about the information about the status of transactions that is deducible from the behavior of $M1_X$. If β is a sequence of actions of $M1_X$, and T and T' are transaction names we say that T is a *local orphan* at X in β if an INFORM_ABORT_AT(X)OF(U) event occurs in β for some ancestor U of T, and we say that T is *lock-visible* at X to T' in β if β contains a subsequence β' consisting of an INFORM_COMMIT_AT(X)OF(U') event for every $U \in ancestors(T) - ancestors(T')$, arranged in ascending order (so the INFORM_COMMIT for $parent(U)$ is preceded by that for U). If β is a behavior of a generic system, we note that T is lock-visible to T' at X in $\beta|M1_X$ only if T is visible to T' in β. Similarly T is a local orphan at X in $\beta|M1_X$ only if T is an orphan in β.

First we have a fundamental invariant of the state of $M1_X$, which expresses the fact that conflicting locks are never held by transactions except when one transaction is the ancestor of the other. This condition is enforced when locks are granted, and preserved thereafter by all actions.

Lemma 9 *Let β be a finite schedule of $M1_X$. Suppose β can leave $M1_X$ in state s, and that $T \in s.write\text{-}lockholders$ and $T' \in s.read\text{-}lockholders \cup s.write\text{-}lockholders$. Then either T is an ancestor of T' or else T' is an ancestor of T.*

The following lemma shows which transactions hold locks after a schedule of $M1_X$.

Lemma 10 *Let β be a finite schedule of $M1_X$. Suppose that β can leave $M1_X$ in state s. Let T be an access to X such that REQUEST_COMMIT(T,v) occurs in β and T is not a local orphan in β, and let T' be the highest ancestor of T such that T is lock-visible to T' in β. If T is a write access then $T' \in s.write\text{-}lockholders$. If T is a read access then $T' \in s.read\text{-}lockholders$.*

The following lemma shows that when an access T' occurs, all prior conflicting accesses must either be local orphans or lock-visible to T'.

Lemma 11 *Let β be a generic object well-formed schedule of $M1_X$. Suppose distinct events π =REQUEST_COMMIT(T,v) and π' =REQUEST_COMMIT(T',v') occur in β, where T and T' conflict. If π precedes π' in β then either T is a local orphan in β' or T is lock-visible to T' in β', where β' is the prefix of β preceding π'.*

The following lemma characterizes the *value* component of the state, showing that $value(T)$ reflects the effects of all transactions that are lock-visible to T.

Lemma 12 *Let β be a finite generic object well-formed schedule of $M1_X$. Suppose that β can leave $M1_X$ in state s. Let T be a transaction name that is not a local orphan in β such that $T \in s.write\text{-}lockholders$. Then $s.value(T) = final\text{-}value(\delta, X)$, where δ is the subsequence of β consisting of events π such that $transaction(\pi)$ is lock-visible to T in β.*

From the previous lemma, we can show a more general characterization.

Lemma 13 *Let β be a finite generic object well-formed schedule of $M1_X$. Suppose that β can leave $M1_X$ in state s. Let T be a transaction name that is not a local orphan in β, and let U denote the least ancestor of T such that $U \in s.write\text{-}lockholders$. Then $s.value(U) = final\text{-}value(\gamma, X)$, where γ is the subsequence of β consisting of events π such that $transaction(\pi)$ is lock-visible to T in β.*

5.4 Correctness Proof of Read/Write Locking

Consider a generic system in which each generic object is $M1_X$ for the appropriate object name X. We will use Theorem 8 to prove that every behavior of this system is serially correct for T_0. The proof relies on first establishing that the system's behaviors have appropriate return values and then showing that the serialization graph is acyclic. We show that the system's behaviors have appropriate return values by showing that REQUEST_COMMIT events for read accesses are current and safe.

Lemma 14 *Let $\mathcal{S}$ be a generic system where for each object name X, $M1_X$ is used as the corresponding generic object automaton. Let β be a finite behavior of $\mathcal{S}$. If π is a REQUEST_COMMIT(T,v) event that occurs in $visible(\beta, T_0)$ where T is a read access to X, then π is current and safe in $serial(\beta)$.*

Proof: Let β' be the prefix of β preceding π and let $\beta'' = \beta'|M1_X$. The preconditions of π and Lemma 13 imply that $v = final\text{-}value(\gamma, S_X)$ where γ is the subsequence of β'' consisting of events whose transaction is lock-visible to T in β''. Thus, to show that π is current in β, it suffices to show that $write\text{-}sequence(\gamma, X) = clean\text{-}write\text{-}sequence(serial(\beta'), X)$.

Since T is not an orphan in β, any transaction lock-visible to T in β'' (and hence visible to T in β) is not an orphan in *serial*(β'). Therefore, *write-sequence*(γ, X) is a subsequence of *clean-write-sequence*(*serial*(β'), X). On the other hand, consider any REQUEST_COMMIT(T',v') event in *clean-write-sequence*(*serial*(β'), X). Then T' is a write access and T' is not an orphan in *serial*(β'); thus T' is not a local orphan in $\beta''\pi$. Since T' conflicts with T, Lemma 11 applied to $\beta''\pi$ implies that T' is lock-visible to T in $\beta''\pi$ and hence in β''. Therefore, REQUEST_COMMIT(T',v') occurs in *write-sequence*(γ, S_X). Thus, *clean-write-sequence*(*serial*(β'), S_X) is a subsequence of *write-sequence*(γ, S_X), so in fact *clean-write-sequence*(*serial*(β'), S_X) = *write-sequence*(γ, S_X). Therefore, π is current in *serial*(β).

If *clean-last-write*(*serial*(β'), S_X) is defined, then Lemma 11 applied to $\beta''\pi$ implies that *clean-last-write*(*serial*(β'), S_X) is lock-visible to T in $\beta''\pi$. Therefore, it is visible to T in *serial*(β'). It follows that π is safe in *serial*(β). □

Proposition 15 *Let $\mathcal{S}$ be a generic system where for each object name X, $M1_X$ is used as the corresponding generic object automaton. Let β be a finite behavior of $\mathcal{S}$. Then serial(β) has appropriate return values.*

Proof: We claim the following:

1. If π is a REQUEST_COMMIT(T,v) event occurring in *visible*(β, T_0), and T is a write access to X, then $v = OK$.

2. If π is a REQUEST_COMMIT(T,v) event occurring in *visible*(β, T_0), and T is a read access to X, then π is current and safe in *serial*(β).

The first of these is immediate, since in the transition relation for each object $M1_X$, $v = OK$ is a precondition on each REQUEST_COMMIT(T,v) action where T is a write access to X. The second follows from Lemma 14. Then the conclusion follows from Lemma 6. □

The following proposition shows that $M1_X$ ensures that the serialization graph is acyclic. The serialization graph consists of two parts, *conflict*(*serial*(β)) and *precedes*(*serial*(β)). The proof shows that each of these is consistent with the completion order; i.e., that if $(U, U') \in$ *conflict*(*serial*(β)), the U completes before U' (and similarly for *precedes*).

Proposition 16 *Let $\mathcal{S}$ be a generic system where for each object name X, $M1_X$ is used as the corresponding generic object automaton. Let β be a finite behavior of $\mathcal{S}$. Then SG(serial(β)) is acyclic.*

Proof: Let T be visible to T_0 in β. We will prove that $SG(serial(\beta), T)$ is acyclic by showing that both *conflict*(*serial*(β)) and *precedes*(*serial*(β)) are subrelations of the partial order *completion*(β), where $(U, U') \in$ *completion*(β) if U and U' are siblings such that either β contains a completion event for U preceding a completion event for U' or β contains a completion event for U and no completion event for U'.

Suppose $(T, T') \in$ *precedes*(*serial*(β)). Then a report event for T and a REQUEST_CREATE(T') occur in *serial*(β), in that order. But there must be a completion event for T preceding the report event; moreover, any completion event for T' must follow the REQUEST_CREATE(T'). It follows that $(T, T') \in$ *completion*(β).

Now suppose that $(T, T') \in$ *conflict*(*serial*(β)). Then there are events ϕ and ϕ' in *visible*(β, T_0) such that $\phi =$ REQUEST_COMMIT(U,v) where U is a descendant of T,

ϕ' = REQUEST_COMMIT(U',v') where U' is a descendant of T', U conflicts with U' and ϕ precedes ϕ' in $visible(\beta, T_0)$. Since U and U' conflict, there is some object name X such that U and U' are both accesses to X. Then $\beta|M1_X$ is a generic object well-formed behavior of $M1_X$ that contains both ϕ and ϕ'. Since $U = transaction(\phi)$ is visible to T_0 in β we know that U is not a local orphan in $\beta|M1_X$. Lemma 11 implies that U is lock-visible to U' in the prefix of $\beta|M1_X$ preceding ϕ'. Since $lca(U, U') = parent(T)$, we see that β contains an INFORM_COMMIT_AT(X)OF(T) event preceding ϕ', and thus (since β is a generic behavior) that a COMMIT(T) event occurs in β preceding ϕ'. On the other hand, U' is live in the prefix of β ending in ϕ', and U' is not an orphan in β (since REQUEST_COMMIT(U',v') occurs in $visible(\beta, T_0)$). Thus T' is live in the prefix of β ending in ϕ' so any completion event for T' in β must follow ϕ' and thus follow the completion event for T. That is, $(T, T') \in completion(\beta)$. □

Now we can prove the main correctness theorem for Moss' algorithm.

Theorem 17 *Let $\mathcal{S}$ be a generic system where for each object name X, $M1_X$ is used as the corresponding generic object automaton. Let β be a finite behavior of $\mathcal{S}$. Then β is serially correct for T_0.*

Proof: Proposition 15 implies that $serial(\beta)$ has appropriate return values. Proposition 16 implies that the graph $SG(serial(\beta))$ is acyclic. Then Theorem 8 implies that β is serially correct for T_0. □

6 Extension to General Data Types

In this section we extend some of the previous results to arbitrary data types. Thus, we allow serial objects to have arbitrary operations, rather than restricting them to be read/write objects.

6.1 Serialization Graphs

In order to define a serialization graph analogously to our previous definitions, we must know how to define "conflict edges," which in turn requires a definition of conflicts between operations of an arbitrary data type. In order to define conflicts, we use two auxiliary definitions, of "equieffectiveness" and "commutativity."

Informally, we say that two finite sequences of external actions of a particular serial object automaton S_X are "equieffective" if they 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 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, $\beta\gamma \in finbehs(S_X)$ if and only if $\beta'\gamma \in finbehs(S_X)$.[9] Obviously, equieffectiveness is a symmetric relation, so that if β is equieffective to β' we often say that β and β' are *equieffective*. Note that if β and β' are serial object well-formed sequences and β is equieffective to β', then if β is in $finbehs(S_X)$, β' must also be in $finbehs(S_X)$.

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 special case is more restrictive than necessary.

[9]This definition first appeared in [5].

We next define a notion of "commutativity" of operations.[10] Let S_X be a serial object for object name X, and let (T, v) and (T', v') be operations, where T and T' are accesses to X. Then we say that (T, v) and (T', v') *commute backwards* provided that for all finite sequences of operations ξ the following holds. If $perform(\xi(T, v)(T', v'))$ is a finite behavior of S_X and both $perform(\xi(T, v)(T', v'))$ and $perform(\xi(T', v')(T, v))$ are serial object well-formed then $perform(\xi(T', v')(T, v))$ is equieffective to $perform(\xi(T, v)(T', v'))$ (and hence is also a behavior of S_X). Note that backward commutativity is a symmetric relation.

We say that two operations (T, v) and (T', v') *conflict* provided that they fail to commute backwards. We say that two accesses T and T' *conflict* provided that there exist v and v' such that (T, v) and (T', v') conflict. We note that the new definition of "conflicts" generalizes the definition given earlier for accesses to a read/write object (where two accesses conflict unless both are read accesses).

The following proposition generalizes Proposition 7, which considered only read/write objects.

Proposition 18 *Suppose that ξ is a sequence of operations of X such that $perform(\xi)$ is a serial object well-formed behavior of S_X. Suppose that η is a reordering of ξ such that all pairs of conflicting operations occur in the same order in η and in ξ. Then $perform(\eta)$ is a behavior of S_X.*

Given the generalized notion of conflict relation defined above and the same notion of precedes used earlier, we define serialization graphs exactly as before. However, we cannot use the same definition of appropriate return values, since it relies on the properties of read/write objects. We generalize it as follows. If β is a simple behavior, we say that β *has appropriate return values* provided that for all object names X, the following is true: $perform(operations(\gamma))$ is a behavior of S_X, where $\gamma = visible(\beta, T_0)|X$. Notice that Lemma 5 shows that this is indeed a generalization of the more concrete definition given for systems where every serial object is a read/write object.

Now we can show our main theorem for arbitrary data types.

Theorem 19 *Let β be a finite simple behavior that has appropriate return values. Suppose that $SG(\beta)$ is acyclic. Then β is serially correct for T_0.*

Proof: The proof is essentially identical to the earlier proof for the read/write case. □

6.2 An Undo Logging Algorithm

Now we use serialization graphs to give a proof of correctness of a particular system, one in which a general "undo logging" algorithm is used everywhere. This algorithm works for objects of arbitrary data type.

We model a system using the undo logging algorithm as a generic system in which every generic object automaton is the "undo logging object automaton" U_X described below. A state s of U_X consists of four components: *s.created*, *s.commit-requested*, *s.committed* and *s.operations*. The first three are sets of transactions, initially empty, and the last is a sequence (log) of operations of X (recording the sequence of operations that have taken place, but with operations removed if they are later found to be aborted), initially the empty sequence. The steps of U_X are as follows:

[10] The definition of commutativity required here is slightly different from the one used in [4]. These definitions and a careful exploration of the differences between them are described in [16].

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:
$s.committed = s'.committed \cup \{T\}$

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Effect:
$s.operations =$
$s'.operations - \{(T', v') | T'$ is a descendant of $T\}$

REQUEST_COMMIT(T,v), T an access to X and v a value
Precondition:
$T \in s'.created - s'.commit\text{-}requested$
(T, v) commutes backward with all (T', v') in
$s'.operations$ such that some U in
$ancestors(T') - ancestors(T)$ is not
in $s'.committed$.
$perform(s'.operations(T, v))$ is a behavior of S_X
Effect:
$s.operations = s'.operations(T, v)$
$s.commit\text{-}requested = s'.commit\text{-}requested \cup \{T\}$

The algorithm is described very abstractly; for example, the "state" is kept simply as a log of operations, rather than in some more compact form. Practical implementations would need to compact the information in the operations log, and restrict the nondeterminism in choosing which active invocation to respond to. Our results apply *a fortiori* to implementations of the algorithm in which the state is compacted, and in which the nondeterminism is restricted.

Informally, the algorithm works as follows. When an operation is executed (i.e., a REQUEST_COMMIT occurs for an access), the operation is appended to *s.operations*. A REQUEST_COMMIT(T,v) is allowed to occur only if it commutes with all operations executed by transactions that are not visible to T. The commit of a transaction is simply recorded in *s.committed*; this component is used in the precondition for REQUEST_COMMIT(T,v) to determine which transactions are visible to T. When a transaction aborts, all operations executed by its descendants are removed from the log; this has the effect of "undoing" all the effects of the transaction.

6.3 Basic Properties of U_X

Here we give some properties of U_X. As before, these can be proved by common techniques such as invariant assertions or arguments about sequences of actions.

The statements of the results below require some terminology describing what can be deduced about the status of transactions from the local behavior of U_X. Let β be a sequence of actions of U_X and let T and T' be transaction names. We define the notion of a local orphan as for $M1_X$: we say that T is a *local orphan* at X in β if an INFORM_ABORT_AT(X)OF(U) event occurs in β for some ancestor U of T. We define a slightly different notion of visibility: we say that T is *locally visible* at X to T' in β if β contains an INFORM_COMMIT_AT(X)OF(U)

event for every $U \in ancestors(T) - ancestors(T')$. (Notice the difference with the definition of *lock-visible*, which requires the INFORM_COMMIT events to occur in leaf-to-root order.) If β is a behavior of a generic system, we note that T is locally visible to T' at X in $\beta|U_X$ only if T is visible to T' in β. Similarly T is a local orphan at X in $\beta|U_X$ only if T is an orphan in β.

The following two lemmas characterize the *operations* component of the state of U_X.

Lemma 20 *Let β be a finite generic object well-formed schedule of U_X that can lead to state s. Then s.operations is exactly the subsequence of operations(β) obtained by removing all operations (T, v) such that an INFORM_ABORT_AT(X)OF(U) for some ancestor U of T occurs after the REQUEST_COMMIT(T,v) in β.*

Lemma 21 *Let β be a finite generic object well-formed schedule of U_X that can lead to state s. Let $\mathcal{T}$ be any set of transaction names such that $\mathcal{T} \cap s.committed = \emptyset$.*

1. *If (T', v') precedes (T'', v'') is s.operations, T' is a descendant of a transaction in $\mathcal{T}$ and T'' is not, then (T', v') commutes backward with (T'', v'').*
2. *If ξ is the sequence of operations obtained by removing the descendants of all transactions in $\mathcal{T}$ from s.operations, then perform(ξ) is a behavior of S_X.*

The next lemma parallels Lemma 11.

Lemma 22 *Let β be a generic object well-formed schedule of U_X. Suppose distinct events π =REQUEST_COMMIT(T,v) and π' =REQUEST_COMMIT(T',v') occur in β, where (T, v) and (T', v') conflict. If π precedes π' in β then either T is a local orphan in β' or T is locally visible to T' in β', where β' is the prefix of β preceding π'.*

6.4 Correctness Proof

First, we show that the condition on appropriate return values is satisfied.

Proposition 23 *Let $\mathcal{S}$ be a generic system where for each object name X, U_X is used as the corresponding generic object automaton. Let β be a finite behavior of $\mathcal{S}$. Then serial(β) has appropriate return values.*

Proof: Fix a particular object name X. We must show that $perform(operations(visible(\beta, T_0)|X))$ is a behavior of S_X. Let s be the unique state of U_X such that β can lead to s. We define $\mathcal{T}$ to be the set of all transactions other than T_0 that are not committed in β. It follows that no transaction in $\mathcal{T}$ can be in *s.committed*.

Lemma 20 implies that *s.operations* is exactly the subsequence of *operations*(β) obtained by removing all operations (T, v) such that an INFORM_ABORT_AT(X)OF(U) for some ancestor U of T occurs after the REQUEST_COM_MIT(T,v) in β. Let ξ be the sequence of operations that results by removing descendants of transactions in $\mathcal{T}$ from *s.operations*. We claim that $operations(visible(\beta, T_0)|X) = \xi$.

The claim is proved as follows: Both sequences are subsequences of *operations*(β), and so common operations occur in the same order. We must show that the same operations appear in both sequences.

Suppose that (T, v) appears in $operations(visible(\beta, T_0)|X)$. Then no ABORT($U$) appears in β for any ancestor U of T, and hence no INFORM_ABORT_AT(X)OF(U) appears in β.

Therefore, (T, v) is in *s.operations*. Also, T cannot be a descendant of any transaction in $\mathcal{T}$, since T is visible to T_0 in β. Therefore, (T, v) appears in ξ.

Now suppose (T, v) appears in ξ. Then T is not a descendant of any transaction in $\mathcal{T}$, so that all ancestors of T except for T_0 are committed in β. Therefore, T is visible to T_0 in β, and so (T, v) appears in $operations(visible(\beta, T_0)|X)$. This establishes the claim.

Now Lemma 21 implies that $operations(visible(\beta, T_0)|X)$ is a behavior of S_X, as needed. □

Next, we show that the serialization graphs are acyclic; the proof of this result is quite similar to that of Proposition 16.

Proposition 24 *Let $\mathcal{S}$ be a generic system where for each object name X, U_X is used as the corresponding generic object automaton. Let β be a finite behavior of $\mathcal{S}$. Then $SG(serial(\beta))$ is acyclic.*

Proof: Let T be visible to T_0 in β. We will prove that $SG(serial(\beta), T)$ is acyclic by showing that both $conflict(serial(\beta))$ and $precedes(serial(\beta))$ are subrelations of the partial order $completion(\beta)$.[11]

Suppose $(T, T') \in precedes(serial(\beta))$. Then a report event for T and a REQUEST_CREATE(T') occur in $serial(\beta)$, in that order. But there must be a completion event for T preceding the report event; moreover, any completion event for T' must follow the REQUEST_CREATE(T'). It follows that $(T, T') \in completion(\beta)$.

Now suppose that $(T, T') \in conflict(serial(\beta))$. Then there are events ϕ and ϕ' in $visible(\beta, T_0)$ such that ϕ = REQUEST_COMMIT(U,v) where U is a descendant of T, ϕ' = REQUEST_COMMIT(U',v') where U' is a descendant of T', U conflicts with U' and ϕ precedes ϕ' in $visible(\beta, T_0)$. Since U and U' conflict, there is some object name X such that U and U' are both accesses to X. Then $\beta|U_X$ is a generic object well-formed behavior of U_X that contains both ϕ and ϕ'. Since $U = transaction(\phi)$ is visible to T_0 in β we know that U is not a local orphan in $\beta|U_X$. Lemma 22 implies that U is locally visible to U' in the prefix of $\beta|U_X$ preceding ϕ'. Since $lca(U, U') = parent(T)$, we see that β contains an INFORM_COMMIT_AT(X)OF(T) event preceding ϕ', and thus (since β is a generic behavior) that a COMMIT(T) event occurs in β preceding ϕ'. On the other hand, U' is live in the prefix of β ending in ϕ', and U' is not an orphan in β (since REQUEST_COMMIT(U',v') occurs in $visible(\beta, T_0)$). Therefore T' is live in the prefix of β ending in ϕ' so any completion event for T' in β must follow ϕ' and thus follow the completion event for T. That is, $(T, T') \in completion(\beta)$. □

Theorem 25 *Let $\mathcal{S}$ be a generic system where for each object name X, U_X is used as the corresponding generic object automaton. Let β be a finite behavior of $\mathcal{S}$. Then β is serially correct for T_0.*

Proof: Proposition 23 implies that $serial(\beta)$ has appropriate return values. Proposition 24 implies that the graph $SG(serial(\beta))$ is acyclic. Then Theorem 19 implies that β is serially correct for T_0. □

7 Conclusions

In this paper we have presented a proof technique for nested transaction systems. Using this technique, two properties must be demonstrated to show correctness: the return values for

[11] Recall that $(U, U') \in completion(\beta)$ if U and U' are siblings such that either β contains a completion event for U preceding a completion event for U' or else β contains a completion event for U and no completion event for U'.

operations must be shown to be "appropriate," and a "serialization graph" must be shown to be acyclic. The first property corresponds to an assumption that is made implicitly in the classical theory of concurrency control. The second property generalizes the serialization graphs of the classical theory to nested transactions.

The classical theory has been extended in a variety of ways, for example to model concurrency control and recovery algorithms that use multiple versions, and to model replication algorithms.[12] It should be possible to develop techniques based on the model presented in this paper that parallel the techniques used in the classical theory for these other kinds of systems.

Acknowledgements

We thank Michael Merritt for many useful comments that helped improve the content and presentation of this paper.

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] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. *Commutativity-Based Locking for Nested Transactions.* Technical Memo MIT/LCS/TM-370.b, Massachusetts Institute Technology, Laboratory for Computer Science, August 1989. To appear in JCSS.

[5] 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.

[6] 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.

[7] T. Hadzilacos and V. Hadzilacos. Transaction synchronisation in object bases. In *7th ACM Symposium on Principles of Database Systems*, pages 193–200, Austin, TX, March 1988.

[8] 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. To appear in Journal of the ACM.

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

[10] N. Lynch and M. Merritt. Introduction to the theory of nested transactions. In *International Conference on Database Theory*, pages 278–305, Rome, Italy, September 1986. Also, expanded version to appear in Theoretical Computer Science.

[12] These extensions to the classical theory have typically required redefining the notion of correctness (e.g., introducing the notion of "1-copy serializability"). In contrast, the definition of correctness used in our work, namely that a system's behaviors must be serially correct for T_0, is sufficiently general to apply directly to these and many other kinds of systems.

[11] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. A theory of atomic transactions. In *International Conference on Database Theory*, Bruges, Belgium, September 1988. Also, available as MIT/LCS/TM-362 June 1988.

[12] 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.

[13] 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.

[14] C. Papadimitriou. *The Theory of Concurrency Control.* Computer Science Press, 1986.

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

[16] W. E. Weihl. The impact of recovery on concurrency control (extended abstract). In *Symposium on Principles of Database Systems*, pages 259–269, Philadelphia, PA, March 1989.

[17] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[18] 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.