4

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-370

COMMUTATIVITY-BASED LOCKING FOR NESTED TRANSACTIONS

Alan Fekete
Nancy Lynch
Michael Merritt
Bill Weihl

August 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-370	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-85-K-0168 and N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
COMMUTATIVITY-BASED LOCKING FOR NESTED TRANSACTIONS

12. PERSONAL AUTHOR(S)
Fekete, Alan; Lynch, Nancy; Merritt, Michael; and Weihl, Bill

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1988 August	74

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	transactions, atomicity, nested transactions, locking, concurrency control, serializability, recovery, I/O automata

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

A new model is introduced for reasoning about atomic transactions. This model allows careful statement of the correctness conditions to be satisfied by transaction-processing algorithms, as well as clear and concise description of such algorithms. It also serves as a framework for rigorous correctness proofs. A new algorithm is introduced for general commutativity-based locking in nested transaction systems. This algorithm and a previously-known read-update locking algorithm are presented and proved correct using the new model. The proofs are based on Serializability Theorem and a new dynamic atomicity condition for data objects.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

Commutativity-Based Locking for Nested Transactions

Alan Fekete
Nancy Lynch
Michael Merritt
Bill Weihl

24 August 1988[1]

Abstract:

A new model is introduced for reasoning about atomic transactions. This model allows careful statement of the correctness conditions to be satisfied by transaction-processing algorithms, as well as clear and concise description of such algorithms. It also serves as a framework for rigorous correctness proofs. A new algorithm is introduced for general commutativity-based locking in nested transaction systems. This algorithm and a previously-known read-update locking algorithm are presented and proved correct using the new model. The proofs are based on a general *Serializability Theorem* and a new *dynamic atomicity* condition for data objects.

1. Introduction

1.1. Atomic Transactions

The abstract notion of "atomic transaction" was originally developed to hide the effects of failures and concurrency in centralized database systems. It has since been generalized to incorporate a nested structure, and has been applied to problems in both centralized and distributed systems.

Roughly speaking, a transaction is a sequence of accesses to data objects; it should execute "as if" it ran with no interruption by other transactions. Moreover, a transaction can complete either successfully or unsuccessfully, by "committing" or "aborting". If it commits, any alterations it makes to the database should be lasting; if it aborts, it should be "as if" it never altered the database at all. The execution of a set of transactions should be "serializable", that is, equivalent to an execution in which no transactions run concurrently and in which all accesses of committed transactions, but no accesses of aborted transactions, are performed.

The original motivation for transactions was to provide a way of maintaining the consistency of a database. Maintaining consistency is difficult because the hardware can fail, and because users can access the database concurrently. Transactions provide fault-tolerance by guaranteeing

---

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

that either all or none of the effects of a transaction occur. Transactions also simplify the problems of concurrent access by synchronizing the access of concurrent users so that the users appear to access the database sequentially. The net effect is that one can guarantee that consistency is preserved by ensuring that each transaction, when run alone and to completion, preserves consistency. Given that each transaction preserves consistency, any serial execution of transactions without failures (i.e., where each transaction runs to completion) also preserves consistency. Since any serializable concurrent execution is equivalent to a serial execution without failures, any serializable concurrent execution also preserves consistency.

Although much of the database literature focuses on preserving consistency, this alone is not enough. Consider, for example, a simple database system in which no transaction ever actually modifies the database. Such a database is always in a consistent state (assuming that the initial state is consistent), but it is not very useful. A useful system should also guarantee something about the connection between different transactions, and between transactions and the database state. For example, ordinary serializability requires the final state of the database to be the same as after a serial execution in which the same transactions occur. The "view serializability" condition insists in addition that accesses to data return the same values as in the equivalent serial execution. Also, either ordinary serializability or view serializability can be augmented by an "external consistency" condition, which requires that the order of transactions in the equivalent serial execution should be compatible with the order in which transaction invocations and responses occur. A discussion of several correctness conditions can be found in Chapter 2 of [23].

Recently, transactions have been explored as a way of organizing programs for distributed systems [16, 25]. Here, their purpose is not just to provide a way of keeping the state of the database consistent, but also to provide the programmer with mechanisms that simplify reasoning about programs. Failures and concurrency make it harder to reason about programs because of the complexity of the interactions among concurrent activities, and because of the multitude of failure modes. (See, for instance, the banking example in [16].) Transactions help here by allowing the programmer to view a complex piece of code as if it is run atomically: it appears to happen instantaneously, and it happens either completely or not at all.

1.2. Nested Transactions

In order for transactions to be useful for general distributed programming, the notion needs to be extended to include nesting. Thus, in addition to accesses, a transaction can also contain subtransactions. The transaction nesting structure can be described by a forest, with the top-level transactions at the roots and the accesses to data at the leaves. (In general, leaves may occur at any level. For example, a top-level transaction might be a single data access, or might invoke a subtransaction and a data access as sibling subtransactions.) The semantics of nested transactions generalize those of ordinary transactions as follows. Each set of sibling transactions or subtransactions is supposed to execute serializably. As with top-level transactions, subtransactions can commit or abort. Each set of sibling transactions runs as if all the transactions that committed ran in a serial order, and all the transactions that aborted did not run at all. An external consistency property is also required for each set of siblings.

Nested transactions provide a very flexible programming mechanism. They allow the programmer to describe more concurrency than would be allowed by single-level transactions,

by having transactions request the creation of concurrent subtransactions. They also allow localized handling of transaction failures. When a subtransaction commits or aborts, the commit or abort is reported to its parent transaction. The parent can then decide on its next action based on the reported results. For example, if a subtransaction aborts, its parent can use the reported abort to trigger another subtransaction, one that implements some alternative action. This flexible mechanism for handling failures is especially useful in distributed systems, where failures are common because of the unreliability of communication.

Nested transactions are useful in other ways in distributed systems. For example, they can be used to implement remote procedure calls with a "zero or once" semantics: the call appears to happen either zero or one times despite retransmissions of request messages caused by poorly chosen timeouts, lost acknowledgements and other problems of unreliable communication. This is accomplished by treating incomplete or redundant calls as aborted subtransactions of the caller, and undoing their activity without aborting the successful call. For another example, nested transactions aid in the construction of replicated systems. The reading and writing of individual copies of data objects can be done as subtransactions; even if some of the copies fail to respond (causing their subtransactions to fail), the overall transaction can still succeed if enough of the copies respond.

The idea of nested transactions seems to have originated in the "spheres of control" work of [6]. Reed [24] developed the current notion of nesting and designed a timestamp-based implementation. Moss [21] later designed a locking implementation that serves as the basis of the implementation of the Argus programming language. A special case of nesting, emphasizing levels of data abstraction, is used in System R and has been studied in [4], [3], [22], [28].

1.3. Transaction-Processing Algorithms

Many algorithms have been proposed and used for implementing non-nested atomic transactions [7, 26, 15] and also for implementing nested transactions [24, 21]. These algorithms make use of various techniques, including some based on locks, timestamps, multiple versions of data objects, and multiple replicas. The most popular algorithms in practice are probably "read-update" locking algorithms such as those in [7, 21], in which transactions must acquire read locks or update locks on data objects in order to access the objects in the corresponding manner. Update locks are defined to conflict with other locks on the same data object, and conflicting locks are not permitted to be held simultaneously. Thus, a transaction that updates a data object prevents or delays the operation of any other transaction that also wishes to update the same object. The recent book [5] provides an excellent survey of the most important transaction-processing algorithms for non-nested transactions.

There is an important limitation on the usefulness of read-update locking. Many systems contain "concurrency bottlenecks": for example, if the data is organized into a hierarchical structure, the roots of the structure are likely to be accessed by most of the transactions. If read-update locking is used, a transaction that modifies the roots will prevent any other transaction from accessing the root until the modifying transaction commits and releases its lock. Thus, most transactions will be blocked for a significant period, and the throughput performance will suffer. Concurrency bottlenecks also occur when the database contains data that summarizes other data, such as a record of the total assets of a bank, In such cases, most transactions that update the database will need to update the summary data, and thus will exclude

one another from concurrent activity if update locks need to be obtained on the summary data.

Concurrency bottlenecks can often be avoided by using a concurrency control protocol specific to a given data type. Read-update locking itself is a simple example of such a protocol: transactions executing read operations can be allowed to run concurrently without sacrificing atomicity. The correctness of this protocol depends on type-specific properties of the transactions, namely, that certain operations do not modify the state of the database. This example can be generalized to allow more concurrency than can be permitted by read-update locking. For example, operations on summary data such as the total assets of a bank often include increment, decrement, and read operations. Increment and decrement operations are executed by transactions that transfer money into or out of the bank. Using read-update locking, transactions executing increment and decrement operations must exclude each other. However, it is possible to design more permissive concurrency control protocols for this example that use the fact that increment and decrement operations commute to allow transactions executing them to run concurrently. (Cf. IMS Fast Path [9].)

1.4. Need for a Formal Model

There are two reasons why a formal model is needed for reasoning about atomic transactions. First, the implementors of languages that contain transactions need a model in order to reason about the correctness of their implementations. Some of the algorithms that have been proposed for implementing transactions are complicated, and informal arguments about their correctness are not convincing. In fact, it is not even obvious how to state the precise correctness conditions to be satisfied by the implementations; a model is needed for describing the semantics of transactions carefully and formally. Second, if programming languages containing transactions become popular, users of these languages will need a model to help them reason about the behavior of their programs.

There has been considerable prior work on models for atomic transactions, summarized in [5]. This "classical" theory is primarily applicable to single-level transactions, rather than nested transactions. It treats both concurrency control and recovery algorithms, although the treatments of the two kinds of algorithms are not completely integrated. The theory assumes a system organization in which accesses are passed from the transactions to a "scheduler", which determines the order in which they are to be performed by the database. The database handles recovery from transaction abort and media failure, so that each access to one data object is performed in the state resulting from all previous non-aborted accesses to that object. The notion of "serializability" in this theory can be described as "looking like a serial execution, from the point of view of the database". Proofs for some algorithms are presented, primarily based on one combinatorial theorem, the "Serializability Theorem". This important basic theorem states that serializability is equivalent to the absence of cycles in a graph representing dependencies among transactions. There has also been some recent work extending some of the ideas of the classical theory to encompass special cases of nested transactions [4, 3, 28].

This work has some limitations. First, the notion of correctness is too restrictive, stated as it is in terms of the object boundary in a particular system organization. The object interface that is described is suitable for single-version locking and timestamp algorithms (in the absence of transaction aborts), but it is much less appropriate for other kinds of algorithms. Multi-version algorithms and replicated data algorithms, for example, maintain object information in a form

that is very different from the (single-copy latest-value) form used for the simple algorithms, and the appropriate object interface is also very different. The correctness conditions presented for the simple algorithms in [5] thus do not apply without change to these other kinds of algorithms. It seems more appropriate, and useful in not unduly restricting possible implementations, to state correctness conditions at the user interface to the system, rather than at the object boundary.

Second, there is no operational model for transactions; instead, they are characterized using axioms about their executions. However, there are many situations in which such an operational model would be useful. For example, it is possible for a transaction to create a subtransaction because of the fact that an earlier subtransaction aborted; an operational model is helpful in capturing this dependence. Also, it is sometimes interesting to describe how the same transaction would behave in different systems. Such reasoning is facilitated by an operational model that clarifies which actions occur under the transaction's control, and which are due to activity of the environment.

A third limitation of the classical theory is that not everything one might wish to describe in order to model nested transactions is modelled explicitly. For example, in a distributed transaction-processing system, the task of performing an access to a data object might actually consist of as many as five distinguishable events: a request by a transaction to perform the access, the invocation of the access at the object, the completion of the access at the object, the decision by the system that the access is to be committed rather than aborted, and a report to the transaction of the results of the access. In reasoning about a nested transaction system, it is often useful to consider these events separately, but all are encompassed by a single indivisible event in the classical model.

The paper [17] contains a first attempt to develop a model for nested transactions that does not have these limitations. That paper contains a complete proof of an exclusive locking algorithm for nested transactions, but the framework used there does not appear to extend easily to treat many other transaction-processing algorithms. Preliminary versions of the ideas in this paper appear in [19, 8],

1.5. Contents of This Paper

In this paper, we define a new model for transactions that avoids the problems described in the previous subsection. This improvement does not come for free: our model contains more detail than the classical model, and may therefore seem more complicated. It seems to us, however, that this extra detail is necessary. In fact, we believe that the extra detail is useful for understanding not just nested transactions, but also ordinary single-level transactions. We use this model to describe the correctness conditions that we use for transaction systems (including nested transaction systems) - notions analogous to "serializability" but stated in terms of the user interface rather than the database.

We then present a new and general locking algorithm that takes advantage of type-specific information, specifically, the commutativity relationships among the operations on data objects, to allow a high degree of concurrency. We also describe the nested transaction read-update locking algorithm of [21] in our framework, and give complete proofs for both algorithms.

The proofs have an interesting structure. First, we prove our "Serializability Theorem", a

general theorem containing a sufficient condition for proving correctness. Although this theorem is more complicated to state than the classical Serializability Theorem, it is similar in spirit: it shows that the existence of a single ordering of transactions that is consistent with the processing of accesses at each object is sufficient to prove serializability. This theorem can be used to prove the correctness of other algorithms besides those in this paper [1].

We next consider a particular system organization, in which the state of the database is maintained by a collection of objects, each performing its own concurrency control and recovery. The form in which the state is maintained and the algorithms used are not constrained. We then define a new condition for data objects, called "dynamic atomicity". The Serializability Theorem is used to show that if all the objects in a system are dynamic atomic, then the system guarantees serializability. We prove that an object using our general commutativity-based locking algorithm satisfies the dynamic atomicity condition and hence that our algorithm guarantees serializability. The proof of the read-update algorithm involves showing that an object using it provides a subset of the behavior of an object using the general algorithm and so also is dynamic atomic.

The rest of the paper is organized as follows. Section 2 contains some mathematical preliminaries. Section 3 contains a brief outline of the I/O automaton model, the basic model for concurrent systems that is used for presenting all of the ideas of this paper. Section 4 contains a description of "serial systems", extremely constrained transaction-processing systems that are defined solely for the purpose of stating correctness conditions for more liberal systems. Section 5 contains a description of "simple systems", very unconstrained transaction-processing systems that represent the common features of most transaction-processing systems. Section 6 contains our Serializability Theorem, stated in terms of simple systems. Section 7 contains a description of "generic systems", in which each data object is represented separately, a definition of the "dynamic atomicity" condition for objects, and a proof (using the Serializability Theorem) that this condition implies correctness. Section 8 contains a description of special restrictions on objects that are exploited in our new algorithm. Section 9 contains the presentation of our general locking algorithm, together with a proof (using dynamic atomicity) of its correctness. Section 10 contains the presentation of Moss's read-update locking algorithm, together with a proof (based on its relationship to the general algorithm) of its correctness. Section 11 contains some final remarks.

2. Mathematical Preliminaries

An *irreflexive partial order* is a binary relation that is irreflexive, antisymmetric and transitive. Two binary relations R and S are *consistent* if their union can be extended to an irreflexive partial order (or in other words, if their union has no cycles).

The formal subject matter of this paper is concerned with finite and infinite sequences describing the executions of automata. Usually, we will be discussing sequences of elements from a universal set of *actions*. Formally, a *sequence* β of actions is a mapping from a prefix of the positive integers to the set of actions. We describe the sequence by listing the images of successive integers under the mapping, writing $\beta = \pi_1\pi_2\pi_3....$[2] Since the same action may occur

[2]We use the symbols β, γ,... for sequences of actions and the symbols π, ϕ and ψ for individual actions.

several times in a sequence, it is convenient to distinguish the different occurrences. Thus, we refer to a particular occurrence of an action in a sequence as an *event*. Formally, an *event* in a sequence $\beta = \pi_1\pi_2...$ of actions is an ordered pair (i,π), where i is a positive integer and π is an action, such that π_i, the i^{th} action in β, is π.

A set of sequences P is *prefix-closed* provided that whenever $\beta \in P$ and γ is a prefix of β,it is also the case that $\gamma \in P$. Similarly, a set of sequences P is *limit-closed* provided that any sequence all of whose finite prefixes are in P is also in P.

3. The Input/Output Automaton Model

In order to reason carefully about complex concurrent systems such as those that implement atomic transactions, it is important to have a simple and clearly-defined formal model for concurrent computation. The model we use for our work is the *input/output automaton* model [20, 18]. This model allows careful and readable descriptions of concurrent algorithms and of the correctness conditions that they are supposed to satisfy. The model can serve as the basis for rigorous proofs that particular algorithms satisfy particular correctness conditions.

This section contains an introduction to a simple special case of the model that is sufficient for use in this paper.[3]

3.1. Basic Definitions

Each system component is modelled as an "I/O automaton", which is a mathematical object somewhat like a traditional finite-state automaton. However, an I/O automaton need not be finite-state, but can have an infinite state set. The actions of an I/O automaton are classified as either "input", "output" or "internal". This classification is a reflection of a distinction in the system being modelled, between events (such as the receipt of a message) that are caused by the environment, events (such as sending a message) that the component can perform when it chooses and that affect the environment, and events (such as changing the value of a local variable) that a component can perform when it chooses, but that are undetectable by the environment except through their effects on later events. In the model, an automaton generates output and internal actions autonomously, and transmits output actions instantaneously to its environment. In contrast, the automaton's input is generated by the environment and transmitted instantaneously to the automaton. Our distinction between input and other actions is fundamental, based on who determines when the action is performed: an automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action.

3.1.1. Action Signatures

A formal description of the classification of an automaton's actions is given by an "action signature". An *action signature* S is an ordered triple consisting of three pairwise-disjoint sets of actions. We write *in*(S), *out*(S) and *int*(S) for the three components of S, and refer to the actions in the three sets as the *input actions*, *output actions* and *internal actions* of S, respectively. We

---

[3]In this paper, we only consider properties of finite executions, and do not consider "liveness" or "fairness" properties.

let *ext*(S) = in(S) ∪ out(S) and refer to the actions in ext(S) as the *external actions* of S. Also, we let *local*(S) = int(S) ∪ out(S), and refer to the actions in local(S) as the *locally-controlled actions* of S. Finally, we let *acts*(S) = in(S) ∪ out(S) ∪ int(S), and refer to the actions in acts(S) as the *actions* of S.

An *external action signature* is an action signature consisting entirely of external actions, that is, having no internal actions. If S is an action signature, then the *external action signature* of S is the action signature *extsig*(S) = (in(S),out(S),∅), i.e., the action signature that is obtained from S by removing the internal actions.

3.1.2. Input/Output Automata

An *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of four components:

- an action signature *sig*(A),
- a set *states*(A) of *states*,
- a nonempty set *start*(A) ⊆ states(A) of *start states*, and
- a transition relation *steps*(A) ⊆ states(A) × acts(sig(A)) × states(A), with the property that for every state s' and input action π there is a transition (s',π,s) in steps(A).[4]

Note that the set of states need not be finite. We refer to an element (s',π,s) of steps(A) as a *step* of A. The step (s',π,s) is called an *input step* of A if π is an input action, and *output steps*, *internal steps*, *external steps* and *locally-controlled steps* are defined analogously. If (s',π,s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled*. The input-enabling property means that the automaton is not able to block input actions. If A is an automaton, we sometimes write *acts*(A) as shorthand for acts(sig(A)), and likewise for in(A), out(A), etc. An I/O automaton A is said to be *closed* if all its actions are locally-controlled, i.e., if in(A) = ∅.

Note that an I/O automaton can be "nondeterministic", by which we mean two things: that more than one locally-controlled action can be enabled in the same state, and that the same action, applied in the same state, can lead to different successor states. This nondeterminism is an important part of the model's descriptive power. Describing algorithms as nondeterministically as possible tends to make results about the algorithms quite general, since many results about nondeterministic algorithms apply *a fortiori* to all algorithms obtained by restricting the nondeterministic choices. Moreover, the use of nondeterminism helps to avoid cluttering algorithm descriptions and proofs with inessential details. Finally, the uncertainties introduced by asynchrony make nondeterminism an intrinsic property of real concurrent systems, and so an important property to capture in our formal model of such systems.

---

[4] I/O automata, as defined in [20], also include a fifth component, an equivalence relation *part*(A) on local(sig(A)). This component is used for describing fair executions, and is not needed for the results described in this paper.

3.1.3. Executions, Schedules and Behaviors

When a system is modelled by an I/O automaton, each possible run of the system is modelled by an "execution", an alternating sequence of states and actions. The possible activity of the system is captured by the set of all possible executions that can be generated by the automaton. However, not all the information contained in an execution is important to a user of the system, nor to an environment in which the system is placed. We believe that what is important about the activity of a system is the externally-visible events, and not the states or internal events. Thus, we focus on the automaton's "behaviors" — the subsequences of its executions consisting of external (i.e., input and output) actions. We regard a system as suitable for a purpose if any possible sequence of externally-visible events has appropriate characteristics. Thus, in the model, we formulate correctness conditions for an I/O automaton in terms of properties of the automaton's behaviors.

Formally, an *execution fragment* of A is a finite sequence $s_0\pi_1s_1\pi_2...\pi_ns_n$ or infinite sequence $s_0\pi_1s_1\pi_2...\pi_ns_n...$ of alternating states and actions of A such that (s_i,π_{i+1},s_{i+1}) is a step of A for every i. An execution fragment beginning with a start state is called an *execution*. We denote the set of executions of A by *execs*(A), and the set of finite executions of A by *finexecs*(A). A state is said to be *reachable* in A if it is the final state of a finite execution of A.

The *schedule* of an execution fragment α of A is the subsequence of α consisting of actions, and is denoted by *sched*(α). We say that β is a *schedule* of A if β is the schedule of an execution of A. We denote the set of schedules of A by *scheds*(A) and the set of finite schedules of A by *finscheds*(A). The *behavior* of a sequence β of actions in acts(A), denoted by *beh*(β), is the subsequence of β consisting of actions in ext(A). The *behavior* of an execution fragment α of A, denoted by *beh*(α), is defined to be beh(sched(α)). We say that β is a *behavior* of A if β is the behavior of an execution of A. We denote the set of behaviors of A by *behs*(A) and the set of finite behaviors of A by *finbehs*(A).

An *extended step* of an automaton A is a triple of the form (s',β,s), where s' and s are in states(A), β is a finite sequence of actions in acts(A), and there is an execution fragment of A having s' as its first state, s as its last state and β as its schedule. (This execution fragment might consist of only a single state, in the case that β is the empty sequence.) If γ is a sequence of actions in ext(A), we say that (s',γ,s) is a *move* of A if there is an extended step (s',β,s) of A such that γ = beh(β).

We say that a finite schedule β of A *can leave* A *in* state s if there is some finite execution α of A with final state s and with sched(α) = β. We say that an action π is *enabled after* a finite schedule β of A if there is a state s such that β can leave A in state s and π is enabled in s.

If α is any sequence of actions and A is an automaton, we write α|A for α|acts(A).

3.2. Composition

Often, a single system can also be viewed as a combination of several component systems interacting with one another. To reflect this in our model, we define an operation called "composition", by which several I/O automata can be combined to yield a single I/O automaton. Our composition operator connects each output action of the component automata with the identically named input actions of any number (usually one) of the other component automata.

In the resulting system, an output action is generated autonomously by one component and is thought of as being instantaneously transmitted to all components having the same action as an input. All such components are passive recipients of the input, and take steps simultaneously with the output step.

3.2.1. Composition of Action Signatures

We first define composition of action signatures. Let I be an index set that is at most countable. A collection $\{S_i\}_{i \in I}$ of action signatures is said to be *strongly compatible*[5] if for all $i, j \in I$, we have

1. $out(S_i) \cap out(S_j) = \emptyset$,
2. $int(S_i) \cap acts(S_j) = \emptyset$, and
3. no action is in $acts(S_i)$ for infinitely many i.

Thus, no action is an output of more than one signature in the collection, and internal actions of any signature do not appear in any other signature in the collection. Moreover, we do not permit actions involving infinitely many component signatures.

The *composition* $S = \Pi_{i \in I} S_i$ of a collection of strongly compatible action signatures $\{S_i\}_{i \in I}$ is defined to be the action signature with

- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$,
- $out(S) = \cup_{i \in I} out(S_i)$, and
- $int(S) = \cup_{i \in I} int(S_i)$.

Thus, output actions are those that are outputs of any of the component signatures, and similarly for internal actions. Input actions are any actions that are inputs to any of the component signatures, but outputs of no component signature.

3.2.2. Composition of Automata

A collection $\{A_i\}_{i \in I}$ of automata is said to be *strongly compatible* if their action signatures are strongly compatible. The *composition* $A = \Pi_{i \in I} A_i$ of a strongly compatible collection of automata $\{A_i\}_{i \in I}$ has the following components:[6]

- $sig(A) = \Pi_{i \in I} sig(A_i)$,
- $states(A) = \Pi_{i \in I} states(A_i)$,
- $start(A) = \Pi_{i \in I} start(A_i)$, and
- $steps(A)$ is the set of triples (s', π, s) such that for all $i \in I$, (a) if $\pi \in acts(A_i)$ then $(s'[i], \pi, s[i]) \in steps(A_i)$, and (b) if $\pi \notin acts(A_i)$ then $s'[i] = s[i]$.[7]

---

[5] A weaker notion called "compatibility" is defined in [20], consisting of the first two of the three given properties only. For the purposes of this paper, only the stronger notion will be required.

[6] Note that the second and third components listed are just ordinary Cartesian products, while the first component uses a previous definition.

[7] We use the notation s[i] to denote the i^{th} component of the state vector s.

Since the automata A_i are input-enabled, so is their composition, and hence their composition is an automaton. Each step of the composition automaton consists of all the automata that have a particular action in their action signature performing that action concurrently, while the automata that do not have that action in their signature do nothing. We will often refer to an automaton formed by composition as a "system" of automata.

If $\alpha = s_0\pi_1s_1...$ is an execution of A, let $\alpha|A_i$ be the sequence obtained by deleting $\pi_j s_j$ when π_j is not an action of A_i, and replacing the remaining s_j by $s_j[i]$. Recall that we have previously defined a projection operator for action sequences. The two projection operators are related in the obvious way: $sched(\alpha|A_i) = sched(\alpha)|A_i$, and similarly $beh(\alpha|A_i) = beh(\alpha)|A_i$.

In the course of our discussions we will often reason about automata without specifying their internal actions. To avoid tedious arguments about compatibility, henceforth we assume that unspecified internal actions of any automaton are unique to that automaton, and do not occur as internal or external actions of any of the other automata we discuss.

All of the systems that we will use for modelling transactions are closed systems, that is, each operation is an output of some component. Also, each output of a component will be an input of at most one other component.

3.2.3. Properties of Systems of Automata

Here we give basic results relating executions, schedules and behaviors of a system of automata to those of the automata being composed. The first result says that the projections of executions of a system onto the components are executions of the components, and similarly for schedules, etc.

Proposition 1: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i\in I}A_i$. If $\alpha \in execs(A)$ then $\alpha|A_i \in execs(A_i)$ for all $i \in I$. Moreover, the same result holds for finexecs, scheds, finscheds, behs and finbehs in place of execs.

Certain converses of the preceding proposition are also true. In particular, we can prove that schedules of component automata can be "patched together" to form a schedule of the composition, and similarly for behaviors.

Proposition 2: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i\in I}A_i$.

1. Let β be a sequence of actions in acts(A). If $\beta|A_i \in scheds(A_i)$ for all $i \in I$, then $\beta \in scheds(A)$.
2. Let β be a finite sequence of actions in acts(A). If $\beta|A_i \in finscheds(A_i)$ for all $i \in I$, then $\beta \in finscheds(A)$.
3. Let β be a sequence of actions in ext(A). If $\beta|A_i \in behs(A_i)$ for all $i \in I$, then $\beta \in behs(A)$.
4. Let β be a finite sequence of actions in ext(A). If $\beta|A_i \in finbehs(A_i)$ for all $i \in I$, then $\beta \in finbehs(A)$.

The preceding proposition is useful in proving that a sequence of actions is a behavior of a composition A: it suffices to show that the sequence's projections are behaviors of the

components of A and then to appeal to Proposition 2.

3.3. Implementation

We define a notion of "implementation" of one automaton by another. Let A and B be automata with the same external action signature, i.e., with extsig(A) = extsig(B). 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. One can also show that if A implements B, then replacing B by A in any system yields a new system in which all finite behaviors are behaviors of the original system.[8]

In order to show that one automaton implements another, it is often useful to demonstrate a correspondence between states of the two automata. Such a correspondence can often be expressed in the form of a kind of abstraction mapping that we call a "possibilities mapping", defined as follows. Suppose A and B are automata with the same external action signature, and suppose f is a mapping from states(A) to the power set of states(B). That is, if s is a state of A, f(s) is a set of states of B. The mapping f is said to be a *possibilities mapping* from A to B if the following conditions hold:

1. For every start state s_0 of A, there is a start state t_0 of B such that $t_0 \in f(s_0)$.
2. Let s' be a reachable state of A, $t' \in f(s')$ a reachable state of B, and (s',π,s) a step of A. Then there is an extended step, (t',γ,t), of B (possibly having an empty schedule) such that the following conditions are satisfied:
   a. $\gamma|ext(B) = \pi|ext(A)$, and
   b. $t \in f(s)$.

Proposition 3: Suppose that A and B are automata with the same external action signature and there is a possibilities mapping, f, from A to B. Then A implements B.

3.4. Preserving Properties

Although automata in our model are unable to block input actions, it is often convenient to restrict attention to those behaviors in which the environment provides inputs in a "sensible" way, that is, where the environment obeys certain "well-formedness" restrictions. A useful way of discussing such restrictions is in terms of the notion that an automaton "preserves" a property of behaviors: as long as the environment does not violate the property, neither does the automaton. Such a notion is primarily interesting for properties that are prefix-closed and limit-closed. Let Φ be a set of actions and P be a nonempty, prefix-closed, limit-closed set of sequences of actions in Φ (i.e., a nonempty, prefix-closed, limit-closed "property" of such sequences). Let A be an automaton with $\Phi \subseteq ext(A)$. We say that A *preserves* P if $\beta\pi \in$ finbehs(A), $\pi \in$ out(A) and $\beta|\Phi \in P$ together imply that $\beta\pi|\Phi \in P$.

Thus, if an automaton preserves a property P, the automaton is not the first to violate P: as long

---

[8] A stronger, and often useful notion of "A implements B" would require both finite ***and infinite*** behaviors of A to be behaviors of B, bchs(A) $\subseteq$ bchs(B). This condition is too strong for us to use in defining correctness conditions for the locking algorithms considered in this paper.

as the environment only provides inputs such that the cumulative behavior satisfies P, the automaton will only perform outputs such that the cumulative behavior satisfies P. Note that the fact that an automaton A preserves a property P does not imply that all of A's behaviors, when restricted to Φ, satisfy P; it is possible for a behavior of A to fail to satisfy P, if an input causes a violation of P. However, the following proposition gives a way to deduce that all of a system's behaviors satisfy P. The lemma says that, under certain conditions, if all components of a system preserve P, then all the behaviors of the composition satisfy P.

Proposition 4: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and suppose that A, the composition, is a closed system. Let $\Phi \subseteq ext(A)$, and let P be a nonempty, prefix-closed, limit-closed set of sequences of actions in Φ. Suppose that for each $i \in I$, one of the following is true.

1. $\Phi \subseteq ext(A_i)$ and A_i preserves P, or
2. $\Phi \cap ext(A_i) = \emptyset$.

If $\beta \in behs(A)$, then $\beta|\Phi \in P$.

4. Serial Systems and Correctness

In this section, we develop the formal machinery needed to define correctness for transaction-processing systems. Correctness is expressed in terms of a particular kind of system called a "serial system". We define serial systems here, using I/O automata.

4.1. Overview

Transaction-processing systems consist of user-provided transaction code, plus transaction-processing algorithms designed to coordinate the activities of different transactions. The transactions are written by application programmers in a suitable programming language. Transactions are permitted to invoke operations on data objects. In addition, if nesting is allowed, then transactions can invoke subtransactions and receive responses from the subtransactions describing the results of their processing.

In a transaction-processing system, the transaction-processing algorithms interact with the transactions, making decisions about when to schedule the creation of subtransactions and the performance of operations on objects. In order to carry out such scheduling, the transaction-processing algorithms may manipulate locks, multiple copies of objects, and other data structures. In the system organization emphasized by the classical theory, the transaction processing algorithms are divided into a "scheduler algorithm" and a "database" of objects. The scheduler has the power to decide when operations are to be performed on the objects in the database, but not to perform more complex manipulations on objects (such as maintaining multiple copies). Although this organization is popular, it does not encompass all the useful system designs.

In this paper, each component of a transaction-processing system is described as an I/O automaton. In particular, each transaction is an automaton, and all the transaction-processing algorithms together comprise another automaton. Sometimes, as when describing serial systems or explaining our algorithms, we will use a more detailed structure, and present the transaction-processing algorithms as a composition of a collection of automata, one representing each object, and one representing the rest of the system.

It is not obvious how one ought to model the nested structure of transactions within the I/O automaton model. One might consider defining special kinds of automata that have a nested structure. However, it appears that the cleanest way to model this structure is to describe each subtransaction in the transaction nesting structure as a separate automaton. If a parent transaction T wishes to invoke a child transaction T', T will issue an output action that "requests that T' be created". The transaction-processing algorithms receive this request, and at some later time might decide to issue an action that is an input to the child T' and corresponds to the "creation" of T'. Thus, the different transactions in the nesting structure comprise a forest of automata, communicating with each other indirectly through the transaction-processing automaton. The highest-level user-defined transactions, i.e., those that are not subtransactions of any other user-defined transactions, are the roots in this forest.

It is actually more convenient to model the transaction nesting structure as a tree rather than as a forest. Thus, we add an extra "root" automaton as a "dummy transaction", located at the top of the transaction nesting structure. The highest-level user-defined transactions are considered to be children of this new root. The root can be thought of as modelling the outside world, from which invocations of top-level transactions originate and to which reports about the results of such transactions are sent; indeed, we will generally regard the boundary between this root transaction and the rest of the system as the "user interface" to the system. The use of the root transaction works out nicely in the formal development: in most cases, the reasoning we do about this dummy root transaction is very similar to the reasoning we do about ordinary transactions, so that regarding the root as a transaction leads to economy in our formal arguments.

The main purpose of this section is to define correctness conditions to be satisfied by transaction-processing systems. In general, correctness conditions for systems composed of I/O automata are stated in terms of properties of sequences of external actions, and we will follow that convention in this paper. Here it seems most natural to define correctness conditions in terms of the actions occurring at the boundary between the transactions (including the dummy root transaction) and the transaction-processing automaton. For it is immaterial how the transaction-processing algorithms work, as long as the outside world and the transactions see "correct" behavior.

We define correct behavior for a transaction-processing system in terms of the behavior of a particular and very constrained transaction-processing system, one that processes all transactions serially. We call such a system a "serial system". Serial systems consist of transaction automata and "serial object automata" composed with a "serial scheduler automaton". Transaction automata have already been mentioned above. Serial object automata serve as specifications for permissible object behavior. They describe the responses the objects should make to arbitrary sequences of operation invocations, assuming that later invocations wait for responses to previous invocations. Serial objects are very much like the ordinary typed variables that occur in sequential programming languages.

The serial scheduler handles the communication among the transactions and serial objects, and thereby controls the order in which the transactions 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.

We do not consider serial systems to be useful transaction-processing systems to implement, because they allow no concurrency among sibling transactions, and have only a very limited ability to cope with transaction failures. Rather, we use them exclusively as specifications for correct behavior of other, more interesting systems. In later sections, we will describe some systems that do allow concurrency and recovery from transaction failures. (For example, they undo the effects of aborted transactions that have performed significant activity.) We prove that these systems are correct in the sense that certain transactions, in particular the root transaction, are unable to distinguish these systems from corresponding serial systems. It appears to the transactions as if all siblings are run serially, and aborted transactions are never created.

In the remainder of this section, we develop all the necessary machinery for defining serial systems. First, we define a type structure used to name transactions and objects. Then we describe the general structure of a serial system — the components it includes, the actions the components perform, and the way that the components are interconnected. We define several concepts involving the actions of a serial system. We then define the components of the serial system in detail, and state some basic properties of serial systems. Next, we use serial systems to state the correctness conditions that we will use for the remainder of this paper.

4.2. System Types

We begin by defining a type structure that will be used to name the transactions and objects in a serial system.

A *system type* consists of the following:

- a set $\mathcal{T}$ of *transaction names*,
- a distinguished transaction name $T_0 \in \mathcal{T}$,
- a subset *accesses* of $\mathcal{T}$ not containing T_0,
- a mapping *parent*: $\mathcal{T} - \{T_0\} \rightarrow \mathcal{T}$, which configures the set of transaction names into a tree, with T_0 as the root and the accesses as the leaves,
- a set $\mathcal{X}$ of *object names*,
- a mapping *object*: accesses $\rightarrow \mathcal{X}$, and
- a set V of *return values*.

Each element of the set "accesses" is called an *access* transaction name, or simply an *access*. Also, the set of accesses T for which object(T) = X is called $accesses_X$; if $T \in accesses_X$ we say that T is an *access* to X.

In referring to the transaction tree, we use standard tree terminology, such as "leaf node", "internal node", "child", "ancestor", and "descendant". As a special case, we consider any node to be its own ancestor and its own descendant, i.e. the "ancestor" and "descendant" relations are reflexive. We also use the notion of a "least common ancestor" of two nodes.

The transaction tree describes the nesting structure for transaction names, with T_0 as the name of the dummy "root transaction". Each child node in this tree represents the name of a subtransaction of the transaction named by its parent. The children of T_0 represent names of the top-level user-defined transactions. The accesses represent names for the lowest-level transactions in the transaction nesting structure; we will use these lowest-level transactions to model operations on data objects. Thus, the only transactions that actually access data are the leaves of the transaction tree and these do nothing else. The internal nodes model transactions whose function is to create and manage subtransactions including accesses, but they do not access data directly.

The tree structure should 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 an infinite structure with infinite branching.

Classical concurrency control theory, as represented, for example, in [5], considers transactions having a simple nesting structure. As modelled in our framework, that nesting structure has three levels; the top level consists of the root T_0, modelling the outside world, the next level consists of all the user-defined transactions, and the lowest level consists of the accesses to data objects.

The set $\mathcal{X}$ is the set of names for the objects used in the system. Each access transaction name is assumed to be an access to some particular object, as designated by the "object" mapping. The set V of return values is the set of possible values that might be returned by successfully-completed transactions to their parent transactions.

If T is an access transaction name, and v is a return value, we say that the pair (T,v) is an *operation* of the given system type. Thus, an operation includes a designation of a particular access to an object, together with a designation of the value returned by the access.

4.3. General Structure of Serial Systems

A serial system for a given system type is a closed system 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 a single "serial scheduler automaton". Later in this section, we will give a precise definition for the serial scheduler automaton, and will give conditions to be satisfied by the transaction and object automata. Here, we just describe the signatures of the various automata, in order to explain how the automata are interconnected.

The following diagram depicts the structure of a serial system.

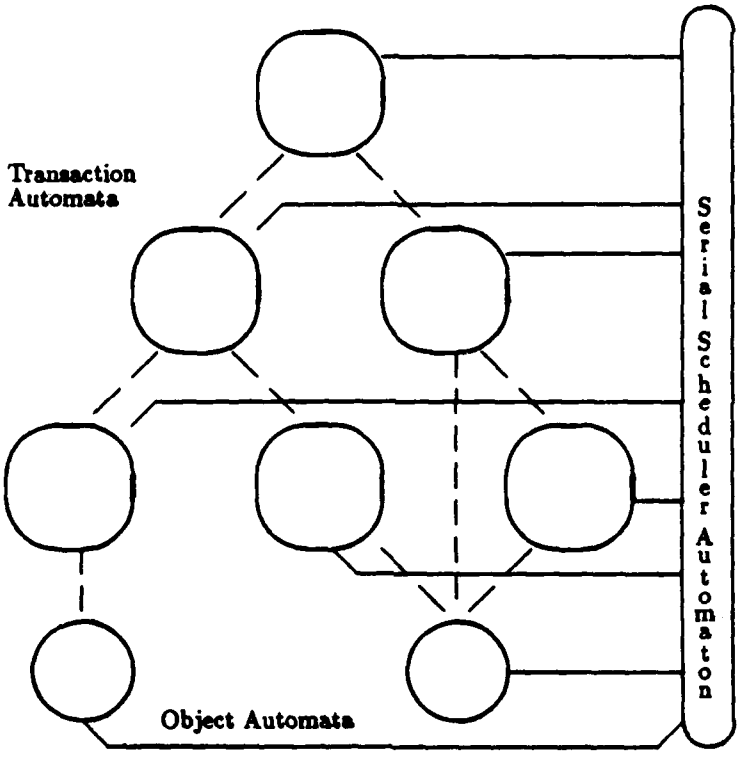

Figure 1: Serial System Structure

The transaction nesting structure is indicated by dotted lines between transaction automata corresponding to parent and child, and between each serial object automaton and the transaction automata corresponding to parents of accesses to the object. The direct connections between automata (via shared actions) are indicated by solid lines. Thus, the transaction automata interact directly with the serial scheduler, but not directly with each other or with the object automata. The object automata also interact directly with the serial scheduler.

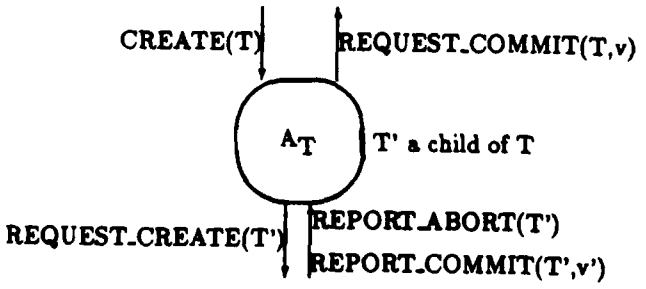

Figure 2: Transaction Automaton

Figure 2 shows the interface of a transaction automaton in more detail. Transaction T has an input CREATE(T) action, which is generated by the serial scheduler in order to initiate T's processing. We do not include explicit arguments to a transaction in our model; rather we suppose that there is a different transaction for each possible set of arguments, and so any input to the transaction is encoded in the name of the transaction. T has REQUEST_CREATE(T') actions for each child T' of T in the transaction nesting structure; these are requests for creation of child transactions, and are communicated directly to the serial scheduler. At some later time, the scheduler might respond to a REQUEST_CREATE(T') action by issuing a CREATE(T') action, an input to transaction T'. T also has REPORT_COMMIT(T',v) and REPORT_ABORT(T') input actions, by which the serial scheduler informs T about the fate (commit or abort) of its previously-requested child T'. In the case of a commit, the report includes a return value v that provides information about the activity of T'; in the case of an abort, no information is returned. Finally, T has a REQUEST_COMMIT(T,v) output action, by which it announces to the scheduler that it has completed its activity successfully, with a particular result as described by return value v.

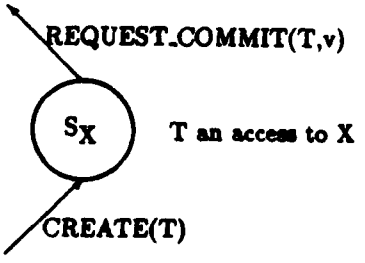

Figure 3: Object Automaton

Figure 3 shows the object interface. Object X has input CREATE(T) actions for each T in $accesses_X$. These actions should be thought of as invocations of operations on object X. Object X also has output actions of the form REQUEST_COMMIT(T,v), representing responses to the invocations. The value v in a REQUEST_COMMIT(T,v) action is a return value returned by the object as part of its response. We have chosen to use the "create" and "request_commit" notation for the object actions, rather than the more familiar "invoke" and "respond" terminology, in the

interests of uniformity: there are many places in our formal arguments where access transactions can be treated uniformly with non-access transactions, and so it is useful to have a common notation for them.

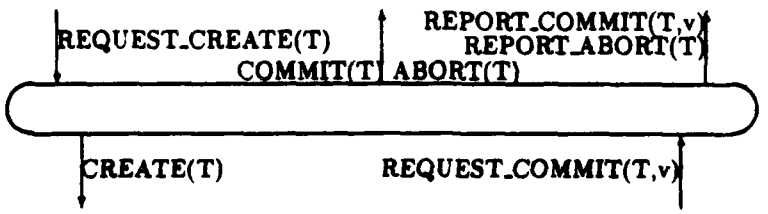

Figure 4: Serial Scheduler Automaton

Figure 4 shows the serial scheduler interface. The serial scheduler receives the previously-mentioned REQUEST_CREATE and REQUEST_COMMIT actions as inputs from the other system components. It produces CREATE actions as outputs, thereby awakening transaction automata or invoking operations on objects. It also produces COMMIT(T) and ABORT(T) actions for arbitrary transactions $T \neq T_0$, representing decisions about whether the designated transaction commits or aborts. For technical convenience, we classify the COMMIT and ABORT actions as output actions of the serial scheduler, even when they are not inputs to any other system component.[8] Finally, the serial scheduler has REPORT_COMMIT and REPORT_ABORT actions as outputs, by which it communicates the fates of transactions to their parents.

As is always the case for I/O automata, the components of a system are determined statically. Even though we referred earlier to the action of "creating" a child transaction, the model treats the child transaction as if it had been there all along. The CREATE action is treated formally as an input action to the child transaction; the child transaction will be constrained not to perform any output actions until such a CREATE action occurs. A consequence of this method of modelling dynamic creation of transactions is that the system must include automata for all possible transactions that might ever be created, in any execution. In most interesting cases, this means that the system will include infinitely many transaction automata.

4.4. Serial Actions

The *serial actions* for a given system type are defined to be the external actions of a serial system of that type. These are just the actions listed in the preceding subsection: CREATE(T) and REQUEST_COMMIT(T,v), where T is any transaction name and v is a return value, and REQUEST_CREATE(T), COMMIT(T), ABORT(T), REPORT_COMMIT(T,v), and

[8] Classifying actions as outputs even though they are not inputs to any other system component is permissible in the I/O automaton model. In this case, it would also be possible to classify these two actions as internal actions of the serial scheduler, but then the statements and proofs of the ensuing results would be slightly more complicated.

REPORT_ABORT(T) where $T \neq T_0$ is a transaction name and v is a return value.[9]

In this subsection, we define some basic concepts involving serial actions. All the definitions in this subsection are based on the set of serial actions only, and not on the specific automata in the serial system. For this reason, we present these definitions here, before going on (in the next subsection) to give more information about the system components.

We first present some basic definitions, and then we define "well-formedness" for sequences of external actions of transactions and objects.

4.4.1. Basic Definitions

The COMMIT(T) and ABORT(T) actions are called *completion* actions for T, while the REPORT_COMMIT(T,v) and REPORT_ABORT(T) actions are called *report* actions for T.

With each serial action π that appears in the interface of a transaction or object automaton (that is, with any non-completion action), we associate the natural transaction. Let T be any transaction name. If π is one of the serial actions CREATE(T), REQUEST_COMMIT(T,v), or REQUEST_CREATE(T'), REPORT_COMMIT(T',v') or REPORT_ABORT(T'), where T' is a child of T, then we define *transaction*(π) to be T. If π is a completion action, then transaction(π) is undefined. In some contexts, we will need to associate a transaction with completion actions as well as other serial actions; since a completion action for T can be thought of as occurring "in between" T and parent(T), we will sometimes want to associate T and sometimes parent(T) with the action. Thus, we extend the "transaction(π)" definition in two different ways. 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 parent(T), if π is a completion action for 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. In particular, hightransaction(π) = lowtransaction(π) = transaction(π) for all serial actions other than completion actions.

We also require notation for the object associated with any serial action whose transaction is an access. 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.

We extend the preceding notation to events as well as actions. For example, if π is an event, then we write *transaction*(π) to denote the transaction of the action of which π is an occurrence. We extend the definitions of "hightransaction", "lowtransaction", and "object" similarly. We will extend other notation in this paper in the same way, without further explanation.

Recall that an *operation* is a pair (T,v), consisting of a transaction name and a return value. We can associate operations with a sequence of serial actions: if β is a sequence of serial actions, we say that the operation (T,v) *occurs* in β if there is a REQUEST_COMMIT(T,v) event in β. Conversely, we can associate serial actions with a sequence of operations: for any operation (T,v), let *perform*(T,v) denote the two-action sequence CREATE(T) REQUEST_COMMIT(T,v),

---

[9]Later in the paper, we will define other kinds of systems besides serial systems, namely, simple systems and generic systems. These will also include the serial actions among their external actions; we will still refer to these actions as "serial actions" even though they appear in non-serial systems.

the expansion of (T,v) into its two parts. This definition is extended to sequences of operations in the natural way: if ξ is a sequence of operations of the form ξ'(T,v), then perform(ξ) = perform(ξ') perform(T,v). Thus, the "perform" function expands a sequence of operations into a corresponding alternating sequence of CREATE and REQUEST_COMMIT actions.

Now we require terminology to describe the status of a transaction during execution. Let β be a sequence of serial actions. A transaction name T is said to be *active* in β provided that β contains a CREATE(T) event but no REQUEST_COMMIT event for T. Similarly, T is said to be *live* in β provided that β contains a CREATE(T) event but no completion event for T. (However, note that β may contain a REQUEST_COMMIT for T.) Also, T is said to be an *orphan* in β if there is an ABORT(U) action in β for some ancestor U of T.

We have already used projection operators to restrict action sequences to particular sets of actions, and to actions of particular automata. We now introduce another projection operator, this time to sets of transaction names. Namely, if β is a sequence of serial actions and $\mathcal{U}$ is a set of transaction names, then $\beta|\mathcal{U}$ is defined to be the sequence $\beta|\{\pi$: transaction(π) $\in \mathcal{U}\}$. If T is a transaction name, we sometimes write $\beta|$T as shorthand for $\beta|\{$T$\}$. Similarly, if β is a sequence of serial actions and X is an object name, we sometimes write $\beta|$X to denote $\beta|\{\pi$: object(π) = X$\}$.

Sometimes we will want to use definitions from this subsection for sequences of actions chosen from some other set besides the set of serial actions — usually, a set containing the set of serial actions. We extend the appropriate definitions of this subsection to such sequences by applying them to the subsequences consisting of serial actions. Thus, if β is a sequence of actions chosen from a set Φ of actions, define *serial*(β) to be the subsequence of β consisting of serial actions. Then we say that operation (T,v) *occurs* in β exactly if it occurs in serial(β). A transaction T is said to be *active* in β provided that it is active in serial(β), and similarly for the "live" and "orphan" definitions. Also, $\beta|\mathcal{U}$ is defined to be serial(β)$|\mathcal{U}$, and similarly for restriction to an object.

4.4.2. Well-Formedness

We will place very few constraints on the transaction automata and serial object automata in our definition of a serial system. However, we will want to assume that certain simple properties are guaranteed; for example, a transaction should not take steps until it has been created, and an object should not respond to an operation that has not been invoked. Such requirements are captured by "well-formedness conditions", basic properties of sequences of external actions of the transaction and serial object components. We define those conditions here.

First, we define "transaction well-formedness". Let T be any transaction name. A sequence β of serial actions π with transaction(π) = T is defined to be *transaction well-formed* for T provided the following conditions hold.

1. The first event in β, if any, is a CREATE(T) event, and there are no other CREATE events.
2. There is at most one REQUEST_CREATE(T') event in β for each child T' of T.
3. Any report event for a child T' of T is preceded by REQUEST_CREATE(T') in β.
4. There is at most one report event in β for each child T' of T.

5. If a REQUEST_COMMIT event for T occurs in β, then it is preceded by a report event for each child T' of T for which there is a REQUEST_CREATE(T') in β.

6. If a REQUEST_COMMIT event for T occurs in β, then it is the last event in β.

In particular, if T is an access transaction name, then the only sequences that are transaction well-formed for T are the prefixes of the two-event sequence CREATE(T)REQUEST_COMMIT(T,v). For any T, it is easy to see that the set of transaction well-formed sequences for T is nonempty, prefix-closed and limit-closed.

It is helpful to have an equivalent form of the "transaction well-formedness" definition for use in later proofs.

Lemma 5: Let $\beta\pi$ be a finite sequence of actions ϕ with transaction(ϕ) = T, where π is a single event. Then $\beta\pi$ is transaction well-formed for T exactly if β is transaction well-formed for T and the following conditions hold.

1. If π is CREATE(T), then
   a. there is no CREATE(T) event in β.
2. If π is REQUEST_CREATE(T') for a child T' of T, then
   a. there is no REQUEST_CREATE(T') event in β,
   b. CREATE(T) appears in β, and
   c. there is no REQUEST_COMMIT event for T in β.
3. If π is a report event for a child T' of T, then
   a. REQUEST_CREATE(T') appears in β, and
   b. there is no report event for T' in β.
4. If π is REQUEST_COMMIT(T,v) for some value v, then
   a. there is a report event in β for every child of T for which there is a REQUEST_CREATE event in β,
   b. CREATE(T) appears in β, and
   c. there is no REQUEST_COMMIT event for T in β.

Now we define "serial object well-formedness". Let X be any object name. A sequence of serial actions π with object(π) = X is defined to be *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 $i \neq j$.

Lemma 6: Suppose β is a sequence of serial actions π with object(π) = X. If β is serial object well-formed for X and T is an access to X, then $\beta|T$ is transaction well-formed for T.

Again, we give an equivalent form of the "serial object well-formedness" definition that will be useful in later proofs.

Lemma 7: Let $\beta\pi$ be a finite sequence of actions ϕ with object(ϕ) = X, where π is a single action. Then $\beta\pi$ is serial object well-formed for X exactly if β is serial object well-formed for X and for every finite prefix $\gamma\pi$ of β, where π is a single action, the

following conditions hold.

1. If π is CREATE(T), then
   a. there is no CREATE(T) event in β, and
   b. there are no active accesses in β.
2. If π is REQUEST_COMMIT(T,v) for a return value v, then
   a. T is active in β.

We also say that a sequence ξ of operations (T,v) with object(T) = X is *serial object well-formed* for X if no two operations in ξ have the same transaction name. Clearly, 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. Also, any serial object well-formed sequence of actions of X is a prefix of perform(ξ) for some serial object well-formed sequence of operations ξ.

4.5. Serial Systems

We are now ready to define "serial systems". Serial systems are composed of transaction automata, serial object automata, and a single serial scheduler automaton. There is one transaction automaton A_T for each non-access transaction name T, and one serial object automaton S_X for each object name X. We describe the three kinds of components in turn.

4.5.1. Transaction Automata

A *transaction automaton* A for a non-access transaction name T of a given system type is an I/O automaton with the following external action signature.

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

In addition, A may have an arbitrary set of internal actions. We require A to preserve transaction well-formedness for T, as defined in Sections and . As discussed earlier, this does not mean that all behaviors of A are transaction well-formed, but it does mean that as long as the environment of A does not violate transaction well-formedness, A will not do so. Except for that requirement, transaction automata can be chosen arbitrarily. Note that if β is a sequence of actions, then $\beta|T = \beta|ext(A)$.

Transaction automata are intended to be general enough to model the transactions defined in any reasonable programming language. Of course, there is still work required in showing how to define appropriate transaction automata for the transactions in any particular language. This correspondence depends on the special features of each language, and we do not describe techniques for establishing such a correspondence in this paper.

4.5.2. Serial Object Automata

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

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

In addition, S may have an arbitrary set of internal actions. We require S to preserve serial object well-formedness for X, as defined in Sections and .

As with transaction automata, serial object automata can be chosen arbitrarily as long as they preserve serial object well-formedness.

Serial object automata are intended to be general enough to model variables of any of the system-provided or user-defined types provided in any reasonable programming language. The "semantic information" about a data object that is used in some concurrency control protocols is obtained from the serial object automaton.

4.5.3. Serial Scheduler

There is a single serial scheduler automaton for each system type. It runs transactions according to a depth-first traversal of the transaction tree, running sets of sibling transactions serially. When two or more sibling transactions are available to run (because their parent has requested their creation), the serial scheduler is free to determine the order in which they run. In addition, the serial scheduler can choose nondeterministically to abort any transaction after its parent has requested its creation, as long as the transaction has not actually been created. In the context of this scheduler, the "semantics" of an ABORT(T) action are that transaction T was never created. The scheduler does not permit any two sibling transactions to be live at the same time, and does not abort any transaction while any of its siblings is active. We now give a formal definition of the serial scheduler automaton.

The action signature of the serial scheduler consists of the following actions, for every transaction name T and return value v.

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

Each state s of the serial scheduler consists of six sets, denoted via record notation: s.create_requested, s.created, s.commit_requested, s.committed, s.aborted and s.reported. The set s.commit_requested is a set of operations. The others are sets of transactions. There is exactly one start state, in which the set create_requested is $\{T_0\}$, and the other sets are empty.

We use the notation s.completed to denote s.committed $\cup$ s.aborted. Thus, s.completed is not an actual variable in the state, but rather a "derived variable" whose value is determined as a function of the actual state variables.

The transition relation of the serial scheduler consists of exactly those triples (s',π,s) satisfying the preconditions and yielding the effects described below, where π is the indicated action. We include in the effects only those conditions on the state s that may change with the action. If a component of s is not mentioned in the effects, it is implicit that the set is the same in s' and s.

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

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

CREATE(T)
Precondition:
$T \in$ s'.create_requested - s'.created
$T \notin$ s'.aborted
siblings(T) $\cap$ s'.created $\subseteq$ s'.completed
Effect:
s.created = s'.created $\cup$ {T}

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

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

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

REPORT_ABORT(T), $T \neq T_0$
Precondition:

$T \in s'.aborted$
$T \notin s'.reported$
Effect:
$s.reported = s'.reported \cup \{T\}$

Thus, the input actions, REQUEST_CREATE and REQUEST_COMMIT, simply result in the request being recorded. A CREATE action can occur only if a corresponding REQUEST_CREATE has occurred and the CREATE has not already occurred. Moreover, it cannot occur if the transaction was previously aborted. The third precondition on the CREATE action says that the serial scheduler does not create a transaction until each of its previously created sibling transactions has completed (i.e., committed or aborted). That is, siblings are run sequentially. A COMMIT action can occur only if it has previously been requested and no completion action has yet occurred for the indicated transaction. An ABORT action can occur only if a corresponding REQUEST_CREATE has occurred and no completion action has yet occurred for the indicated transaction. Moreover, it cannot occur if the transaction was previously created. The third precondition on the ABORT action says that the scheduler does not abort a transaction while there is activity going on on behalf of any of its siblings. That is, aborted transactions are dealt with sequentially with respect to their siblings. The result of a transaction can be reported to its parent at any time after the commit or abort has occurred.

The following lemma describes simple relationships between the state of the serial scheduler and its computational history.

Lemma 8: Let β be a finite schedule of the serial scheduler, and let s be a state such that β can leave the serial scheduler in state s. Then the following conditions are true.

1. $T \in$ s.create_requested exactly if $T = T_0$ or β contains a REQUEST_CREATE(T) event.
2. $T \in$ s.created exactly if β contains a CREATE(T) event.
3. $(T,v) \in$ s.commit_requested exactly if β contains a REQUEST_COMMIT(T,v) event.
4. $T \in$ s.committed exactly if β contains a COMMIT(T) event.
5. $T \in$ s.aborted exactly if β contains an ABORT(T) event.
6. $T \in$ s.reported exactly if β contains a report event for T.
7. $s.committed \cap s.aborted = \emptyset$.
8. $s.reported \subseteq s.committed \cup s.aborted$.

The following lemma gives simple facts about the actions appearing in an arbitrary schedule of the serial scheduler.

Lemma 9: Let β be a schedule of the serial scheduler. Then all of the following hold:

1. If a CREATE(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
2. At most one CREATE(T) event appears in β for each transaction T.
3. If a COMMIT(T) event appears in β, then a REQUEST-COMMIT(T,v) event

precedes it in β for some return value v.

4. If an ABORT(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.

5. If a CREATE(T) or ABORT(T) event appears in β and is preceded by a CREATE(T') event for a sibling T' of T, then it is also preceded by a completion event for T'.

6. At most one completion event appears in β for each transaction.

7. At most one report event appears in β for each transaction.

8. If a REPORT-COMMIT(T,v) event appears in β, then a COMMIT(T) event and a REQUEST_COMMIT(T,v) event precede it in β.

9. If a REPORT-ABORT(T) event appears in β, then an ABORT(T) event precedes it in β.

The final lemma of this subsection says that the serial scheduler preserves the well-formedness properties described earlier.

Lemma 10:

1. Let T be any transaction name. Then the serial scheduler preserves transaction well-formedness for T.

2. Let X be any object name. Then the serial scheduler preserves serial object well-formedness for X.

Proof:

1. Let Φ be the set of serial actions, ϕ, with transaction(ϕ) = T. Suppose $\beta\pi$ is a finite behavior of the serial scheduler, π is an output action of the serial scheduler, and $\beta|\Phi$ is transaction well-formed for T. We must show that $\beta\pi|\Phi$ is transaction well-formed for T. If $\pi \notin \Phi$ the result is immediate, so assume that $\pi \in \Phi$, i.e., that transaction(π) = T.

   We use Lemma 5. We already know that $\beta|\Phi$ is transaction well-formed for T. Since π is an output event, π is either a CREATE(T) event or a REPORT event for a child of T. If π is CREATE(T), then since $\beta\pi$ is a schedule of the serial scheduler, Lemma 9 implies that no CREATE(T) occurs in β. If π is a REPORT event for a child T' of T, then Lemma 9 implies that REQUEST_CREATE(T') occurs in β and no other REPORT for T' occurs in β. Then Lemma 5 implies that $\beta\pi|\Phi$ is transaction well-formed for T.

2. The argument for this case is similar.

4.5.4. Serial Systems, Executions, Schedules and Behaviors

A *serial system* of a give system type is the composition of a strongly compatible set of automata indexed by the union of the set of non-access transaction names, the set of object names and the singleton set {SS} (for "serial scheduler"). Associated with each non-access transaction name T is a transaction automaton A_T for T. Associated with each object name X is a serial object automaton S_X for X. Finally, associated with the name SS is the serial scheduler automaton for the given system type. When the particular serial system is understood from context, we will sometimes use the terms *serial executions*, *serial schedules* and *serial behaviors*

for the system's executions, schedules and behaviors, respectively.

We show that serial behaviors are well-formed for each transaction and object name.

Proposition 11: If β is a serial behavior, then the following conditions hold.

1. For every transaction name T, $\beta|T$ is transaction well-formed for T.
2. For every object name X, $\beta|X$ is serial object well-formed for X.

Proof: For non-access transaction names T, or arbitrary object names X, the result is immediate by Proposition 4, the definitions of transaction and object automata, and Lemma 10.

Suppose that T is an access to X. Since $\beta|X$ is serial object well-formed for X, Lemma 6 implies that $\beta|T$ is transaction well-formed for T.

Unless expressly stated, we hanceforth assume an arbitrary but fixed system type and serial system, with $\{A_T|T$ a non-access in $\mathcal{T}\}$ as the non-access transaction automata, and $\{S_X|X \in \mathcal{X}\}$ as the serial object automata.

4.6. Correctness Conditions

Now that we have defined serial systems, we can use them to define correctness conditions for other transaction-processing systems. It is reasonable to use serial systems in this way because of the particular constraints the serial scheduler imposes on the orders in which transactions and objects can perform steps. We contend that the given constraints correspond precisely to the way nested transaction systems ought to appear to behave; in particular, these constraints yield a natural generalization of the notion of serial execution in classical transaction systems. We arrive at a number of correctness conditions by considering *for which system components* this appearance must be maintained: for the external environment T_0, for all transactions, or for all non-orphan transactions.

To express these correctness conditions we define the notion of "serial correctness" of a sequence of actions for a particular transaction name. 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 $\beta|T = \gamma|T$.[10] (Recall that if T is a non-access, we have $\beta|T = \beta|ext(A_T)$ and $\gamma|T = \gamma|ext(A_T)$). If T is a non-access transaction, serial correctness for T guarantees to implementors of T that their code will encounter only situations that can arise in serial executions.

The principal notion of correctness that we use in our work is that of serial correctness of all finite behaviors for the root transaction name T_0. This says that the "outside world" cannot distinguish between the given system and the serial system. However, many of the algorithms we study satisfy stronger correctness conditions. A fairly strong and possibly interesting correctness condition is the serial correctness of all finite behaviors for all non-access transaction names. Thus, neither the outside world nor any of the individual user transactions can distinguish between the given system and the serial system. Note that the definition of

[10] This condition is analogous to the "view serializability" condition of [29], extended to deal with operations other than reads and writes, and with subtransactions.

implementation relative to all non-access transactions does not require that all the transactions see behavior that is part of the same execution of the serial system; rather, each could see behavior arising in a different execution.

We will also consider intermediate conditions such as serial correctness for all non-orphan transaction names. This condition implies serial correctness for T_0 because the serial scheduler does not have the action ABORT(T_0) in its signature, so T_0 cannot be an orphan. Most of the popular algorithms for concurrency control and recovery, including the locking algorithms in this paper, guarantee serial correctness for all non-orphan transaction names. Our Serializability Theorem gives sufficient conditions for showing that a behavior of a transaction-processing system is serially correct for an arbitrary non-orphan transaction name, and can be used to prove this property for many of these algorithms. The usual algorithms do not guarantee serial correctness for orphans, however; in order to guarantee this as well, the use of a special "orphan management" algorithms is generally required. Such algorithms are described and their correctness proved in [i4].

Note that each correctness condition discussed in this section can be applied to many different kinds of transaction-processing systems. All that is needed is that the system be modelled as an I/O automaton with appropriately named actions.

5. Simple Systems

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. A "simple system" consists of the transaction automata together with a special automaton called the "simple database". Our theorem is stated in terms of simple systems.

Many complicated transaction-processing algorithms can be understood as implementations of the simple system. For example, a system containing separate objects that manage locks and a "controller" that passes information among transactions and objects can be represented in this way, and so our theorem can be used to prove its correctness. The same strategy works for a system containing objects that manage timestamped versions and a controller that issues timestamps to transactions.

Later in this paper, we apply this theorem to show that every behavior of certain locking systems is serially correct for non-orphan transactions.

5.1. Simple Database

There is a single simple database for each system type. The action signature of the simple database is that of the composition of the serial scheduler with the serial objects:

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

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

States of the simple database are the same as for the serial scheduler, and the initial states are also the same. The transition relation is as follows.

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

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

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

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

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

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

REPORT_ABORT(T), $T \neq T_0$
Precondition:
$T \in$ s'.aborted
$T \notin$ s'.reported

Effect:
s.reported = s'.reported ∪ {T}

REQUEST_COMMIT(T,v), T an access
Precondition:
T ∈ s'.created
for all v', (T,v') ∉ s'.commit_requested
Effect:
s.commit_requested = s'.commit_requested ∪ {(T,v)}

The next two lemmas are analogous to those previously given for the serial scheduler.

Lemma 12: Let β be a finite schedule of the simple database, and let s be a state that can result from applying β to the start state. Then the following conditions are true.

1. T ∈ s.create_requested exactly if T = T_0 or β contains a REQUEST_CREATE(T) event.
2. T ∈ s.created exactly if β contains a CREATE(T) event.
3. (T,v) ∈ s.commit_requested exactly if β contains a REQUEST_COMMIT(T,v) event.
4. T ∈ s.committed exactly if β contains a COMMIT(T) event.
5. T ∈ s.aborted exactly if β contains an ABORT(T) event.
6. T ∈ s.reported exactly if β contains a report event for T.
7. s.committed ∩ s.aborted = ∅.
8. s.reported ⊆ s.committed ∪ s.aborted.

Lemma 13: Let β be a schedule of the simple database. Then all of the following hold:

1. If a CREATE(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
2. At most one CREATE(T) event appears in β for each transaction T.
3. If a COMMIT(T) event appears in β, then a REQUEST-COMMIT(T,v) event precedes it in β for some return value v.
4. If an ABORT(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
5. At most one completion event appears in β for each transaction.
6. At most one report event appears in β for each transaction.
7. If a REPORT-COMMIT(T,v) event appears in β, then a COMMIT(T) event and a REQUEST_COMMIT(T,v) event precede it in β.
8. If a REPORT-ABORT(T) event appears in β, then an ABORT(T) event precedes it in β.
9. If T is an access and a REQUEST_COMMIT(T,v) event occurs in β, then a CREATE(T) event precedes it in β.

10. If T is an access, then at most one REQUEST_COMMIT event for T occurs in β.

Proof: By Lemma 12 and the simple database preconditions.

Thus, the simple database embodies those constraints that we would expect any reasonable transaction-processing system to satisfy. The simple database does not allow CREATEs, ABORTs, or COMMITs without an appropriate preceding request, does not allow any transaction to have two creation or completion events, and does not report completion events that never happened. Also, it does not produce responses to accesses that were not invoked, nor does it produce multiple responses to accesses. On the other hand, the simple database allows almost any ordering of transactions, allows concurrent execution of sibling transactions, and allows arbitrary responses to accesses.

We do not claim that the simple database produces only serially correct behaviors; rather, we use the simple database to model features common to more sophisticated systems. Such systems will usually include a controller (perhaps with constraints of its own) and complicated objects with concurrency control and recovery built into them. Such a system will have additional actions for communication between these objects and the controller.

We now show that the simple database preserves transaction well-formedness.

Lemma 14: Let T be any transaction name. Then the simple database preserves transaction-well-formedness for T.

Proof: Let Φ be the set of serial actions, ϕ, with transaction(ϕ) = T. Suppose $\beta\pi$ is a finite behavior of the simple database, π is an output action of the simple database, and $\beta|\Phi$ is transaction-well-formed for T. We must show that $\beta\pi|\Phi$ is transaction-well-formed for T. If $\pi \notin \Phi$, then the result is immediate, so assume that $\pi \in \Phi$, i.e., that transaction(π) = T.

We use Lemma 5. We already know that $\beta|\Phi$ is transaction-well-formed for T. Since π is an output event, π is either a CREATE(T) event for an arbitrary transaction T, a REPORT event for a child of an arbitrary transaction T, or a REQUEST_COMMIT for T, where T is an access. If π is CREATE(T), then since $\beta\pi$ is a schedule of the simple database, Lemma 13 implies that no CREATE(T) occurs in β. If π is a REPORT event for a child T' of T, then Lemma 13 implies that REQUEST_CREATE(T') occurs in β and no other REPORT for T' occurs in β. If π is REQUEST_COMMIT(T,v) and T is an access, then Lemma 13 implies that CREATE(T) occurs in β, and no REQUEST_COMMIT for T occurs in β. Then Lemma 5 implies that $\beta\pi$ is transaction-well-formed for T.

5.2. Simple Systems, Executions, Schedules and Behaviors

A *simple system* is the composition of a compatible set of automata indexed by the union of the set of non-access transaction names and the singleton set {SD} (for "simple database"). Associated with each non-access transaction name T is the transaction automaton A_T for T, and associated with the name SD is the simple database automaton for the given system type. When the particular simple system is understood from context, we will often use the terms *simple executions, simple schedules* and *simple behaviors* for the system's executions, schedules and behaviors, respectively.

Proposition 15: If β is a simple behavior and T is a transaction name, then $\beta|T$ is transaction-well-formed for T.

Proof: The result is immediate by Lemma 14 and the definition of transaction automata.

The following is a basic fact about simple behaviors.

Lemma 16: Let β be a simple behavior. Let T and T' be transaction names, where T' is an ancestor of T. If T is live in β and not an orphan in β then T' is live in β.

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.

6. The Serializability Theorem

In this section, we present our Serializability Theorem, which embodies a fairly general method for proving that a concurrency control algorithm guarantees serial correctness. This theorem expresses the following intuition: a behavior of a system is serially correct provided that there is a way to order the transactions so that when the operations at each object are arranged in the corresponding order, the result is a behavior of the corresponding serial object. The correctness of many different concurrency control algorithms can be proved using this theorem; in this paper, we use it to prove correctness of two locking algorithms.

This theorem is the closest analog we have for the classical Serializability Theorem of [5]. Both that theorem and ours hypothesize that there is some ordering on transactions consistent with the behavior at each object. In both cases, this hypothesis is used to show serial correctness. Our result is somewhat more complicated, however, because it deals with nesting and aborts. In the first two subsections of this section, we give some additional definitions that are needed to accommodate these complications.

6.1. Visibility

One difference between our result and the classical Serializability Theorem is that the conclusion of our result is serial correctness for an arbitrary transaction T, whereas the classical result essentially considers only serial correctness for T_0. Thus, it should not be surprising that the hypothesis of our result does not deal with all the operations at each object, but only with those that are in some sense "visible" to the particular transaction T. In this subsection, we define a notion of "visibility" of one transaction to another. This notion is a technical one, but one that is natural and convenient in the formal statements of results and in their proofs. Visibility is defined so that, in the usual transaction-processing systems, only a transaction T' that is visible to another transaction T can affect the behavior of T.

A transaction T' can affect another transaction T in several ways. First, if T' is an ancestor of T, then T' can affect T by passing information down the transaction tree via invocations. Second, a transaction T' that is not an ancestor of T can affect T through COMMIT actions for T' and all ancestors of T' up to the level of the least common ancestor with T; information can be propagated from T' up to the least common ancestor via COMMIT actions, and from there down to T via invocations. Third, a transaction T' that is not an ancestor of T can affect T by accessing an object that is later accessed by T; in most of the usual transaction-processing

algorithms, this is only allowed to occur if there are intervening COMMIT actions for all ancestors of T' up to the level of the least common ancestor with T.

Thus, we define "visibility" as follows. Let β be any sequence of serial 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.[11]

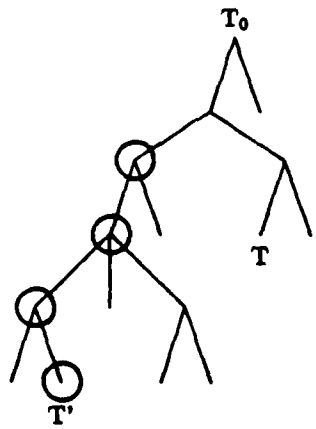

Figure 5: Visibility

Figure 5 depicts two transactions, T and T', neither an ancestor of the other. If the transactions represented by all of the circled nodes have committed in some sequence of serial actions, then the definition implies that T' is visible to T.

The following lemma describes elementary properties of "visibility".

Lemma 17: Let β be a sequence of actions, and let T, T' and T'' be transaction names.

1. If T' is an ancestor of T, then T' is visible to T in β.
2. T' is visible to T in β if and only if T' is visible to lca(T,T') in β.
3. If T'' is visible to T' in β and T' is visible to T in β, then T'' is visible to T in β.
4. If T' is live in β and T' is visible to T in β, then T is a descendant of T'.
5. If T' is an orphan in β and T' is visible to T in β, then T is an orphan in β.

We use the notion of "visibility" to pick, out of a sequence of actions, a subsequence consisting of the actions corresponding to transactions that are visible to a given transaction T. More precisely, 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.

---

[11] Our definition has been chosen for ease of argument — however, note that it says that T' is visible to T even in some situations where T' cannot affect the behavior of T, for example when T' follows T in β.

Note that every action occurring in visible(β,T) is a serial action, even if β itself contains other actions. Note also that the use of "hightransaction" in the definition implies that if T' is visible to T in β and T'' is a child of T' that has an ABORT(T'') in β, then any REQUEST_CREATE(T''), ABORT(T'') and REPORT_ABORT(T'') actions in β are included in visible(β,T), but actions of T'' are not.[12]

The following easy lemma says that the "visible" operator on sequences picks out either all or none of the actions having a particular transaction.

Lemma 18: Let β be a sequence of actions, and let T and T' be transaction names. Then visible(β,T)|T' is equal to β|T' if T' is visible to T in β, and is equal to the empty sequence otherwise.

6.2. Event and Transaction Orders

The hypothesis of the theorem refers to rearranging the operations at each object according to a given order on transactions. The definitions required to describe the appropriate kind of ordering to use for this purpose are provided in this subsection.

6.2.1. Affects Order

We first define a partial order "affects(β)" on the events of a sequence β of serial actions. This will be used to describe basic dependencies between events in a simple behavior; any appropriate ordering will be required to be consistent with these dependencies.

We define the affects relation by first defining a subrelation which we call the "directly-affects" relation and then taking the transitive closure. This decomposition will be useful to us later, when we carry out proofs about the "affects" relation, since it is often easy to reason about "directly-affects". For a sequence β of serial actions, and events φ and π in β, we say that φ *directly affects* π in β (and that (φ,π) ∈ directly-affects(β)) if at least one of the following is true.

- φ and π are serial events with transaction(φ) = transaction(π) and φ precedes π in β,[13]
- 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)
- φ = ABORT(T) and π = REPORT_ABORT(T)

Lemma 19: If β is a simple behavior and (φ,π) ∈ directly-affects(β), then φ precedes π in β.

Proof: The first case is obvious, so we consider only the last five cases of the definition. Transaction-well-formedness implies that there cannot be two REQUEST_CREATE(T) events in β for the same T, and that there cannot be two

[12] If $T = T_0$, visible(β,T) corresponds to the "committed projection" of β as defined in [5].

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

REQUEST_COMMIT events for the same transaction. Also, Lemma 13 says that β does not contain two completion events for the same T. Hence, in each case ϕ is the only occurrence of the appropriate action in β. In each case, π is an output of the simple database, and the simple database preconditions test for the presence of the appropriate preceding action.

For a sequence β of serial actions, define the relation affects(β) to be the transitive closure of the relation directly-affects(β). If the pair (ϕ,π) is in the relation affects(β), we also say that ϕ *affects* π in β. The following is immediate.

Lemma 20: Let β be a simple behavior. Then affects(β) is an irreflexive partial order on the events in β.[14]

Proof: By Lemma 19, ϕ directly affects π in β only if ϕ precedes π in α. Therefore ϕ affects π in β only if ϕ precedes π in β. Thus, affects(β) is irreflexive and antisymmetric. Since affects(β) is constructed as a transitive closure, the result follows.

The conditions listed in the definition of "directly-affects" should seem like a reasonable collection of dependencies among the events in a simple behavior. Here we try to give some technical justification for these conditions. In the proof of the theorem, we will attempt to extract serial behaviors from a given simple behavior. The transaction orderings used to help in this construction will be constrained to be consistent with "affects"; this will mean that the sequences we construct will be closed under "affects" and that the orders of events in these sequences are consistent with "affects". Thus, if β is a simple behavior and $(\phi,\pi) \in$ directly-affects(β), all the serial behaviors we construct that contain π will also contain ϕ, and ϕ will precede π in each such behavior.

The first case of the "directly-affects" definition is used because we are not assuming special knowledge of transaction behavior; if we included π and not ϕ in our candidate serial behavior, we would have no way of proving that the result included correct behaviors of the transaction automata. The remaining cases naturally parallel the preconditions of the serial scheduler; in each case, the preconditions of π as an action of the serial scheduler include a test for a previous occurrence of ϕ, so a sequence of actions with π not preceded by ϕ could not possibly be a serial behavior.

The following lemmas contain some constraints on the kinds of events that can affect other events in a simple behavior.

Lemma 21: Let β be a simple behavior and T a transaction name. Let ϕ and π be events of β such that ϕ affects π in β, lowtransaction(ϕ) is a descendant of T and lowtransaction(π) is not a descendant of T. Then the following hold.

1. Either ϕ is a completion event for T, or ϕ affects a completion event for T that affects π.
2. If no COMMIT event for T appears in β, then ϕ must be an ABORT for T.

Proof: The first case follows from the observation that if ϕ' directly affects π' in β, lowtransaction(ϕ') is a descendant of T and lowtransaction(π') is not a descendant of

[14]Note that the actions of a simple system are exactly the serial actions.

T, then ϕ' is a completion event for T.

The second case follows from the first and the observation that no event directly affects an ABORT event.

The proof of the next lemma is similar.

Lemma 22: Let β be a simple behavior and T a transaction name. Let ϕ and π be events of β such that ϕ affects π in β, lowtransaction(ϕ) is not a descendant of T and lowtransaction(π) is a descendant of T. Then either ϕ is a REQUEST_CREATE(T) event, or ϕ affects a REQUEST_CREATE(T) event for T that affects π.

As before, we extend the "affects" definition to sequences β of arbitrary actions by saying that ϕ affects π in β exactly if ϕ affects π in serial(β).

6.2.2. Sibling Orders

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 orderings that only relate siblings in the transaction nesting tree. We call such an ordering a "sibling order". Interesting examples of sibling orders are the order of completion of transactions or an order determined by assigned timestamps. We define "sibling orders" in this subsection.

Let *SIB* be the (irreflexive) sibling relation among transaction names, for a particular system type; thus, $(T,T') \in SIB$ if and only if $T \neq T'$ and parent(T) = parent(T'). If $R \subseteq SIB$ is an irreflexive partial order then we call R a *sibling order*. Sibling orders are the analog for nested transaction systems of serialization orders in single-level transaction systems. Note that sibling orders are not necessarily total; totality is not always appropriate for our results.

A sibling order can be extended in two natural ways. First, if R is a binary relation on the set of transaction names (such as a sibling order), then let R_{trans} be the extension of R to descendants of siblings, i.e., 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 $(U,U') \in R$. This order echoes the manner in which the serial scheduler runs transactions when it runs siblings with no concurrency, in the order specified by R.[15] Second, if β is any sequence of actions, then $R_{event}(\beta)$ is the extension of R to serial events in β, i.e., 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 $(T,T') \in R_{trans}$. (We use "lowtransaction" in this definition to ensure that completion actions are ordered along with the actions of the completing transaction.) The following are straightforward.

Lemma 23: Let R be a sibling order. Then R_{trans} is an irreflexive partial order, and for any sequence β of actions, $R_{event}(\beta)$ is an irreflexive partial order.

Lemma 24: Let β be a sequence of actions and R a sibling order. Let π and π' be events of β with lowtransactions T and T' respectively. Let ψ and ψ' be events of β with lowtransactions U and U' respectively, where U is a descendant of T and U' is a

[15] A similar definition is used in [4] and [17].

descendant of T'. If $(\pi,\pi') \in R_{event}(\beta)$ then $(\psi,\psi') \in R_{event}(\beta)$.

The concept of a "suitable sibling order" describes two basic conditions that will be required of the sibling orders to be used in our theorem. The first condition is a technical one asserting that R orders sufficiently many siblings, while the second condition asserts that R does not contradict the dependencies described by the affects relation. 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 use of lowtransaction in this definition will ensure that R_{event} imposes an order on ABORT events in visible(β,T).

We have the following extension of the first property above.

Lemma 25: Let β be a simple behavior and T a transaction name. If the sibling order R is suitable for β and T, then R orders all pairs of siblings T' and T'' such that descendants of each are lowtransactions of actions in visible(β,T).

We next give a technical lemma that will be useful for proving that particular sibling orders are suitable.

Lemma 26: Let β be a simple behavior and let R be a sibling order satisfying the following condition. If $(\pi,\pi') \in$ affects(β) 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 affects(β) are consistent partial orders on the events of β.

Proof: We prove this lemma by contradiction. If $R_{event}(\beta)$ and affects(β) are not consistent, then there is a cycle in the relation $R_{event}(\beta) \cup$ affects(β), and thus there must be some shortest cycle. Let $\pi_0, \pi_1, \pi_2, ..., \pi_{n-1}, \pi_n = \pi_0$ be such a shortest cycle, where for each i, $(\pi_i,\pi_{i+1}) \in R_{event}(\beta) \cup$ affects(β). In the following discussion we will use arithmetic modulo n for subscripts, so that if $i = n$, π_{i+1} is to be interpreted as π_1. We note that $n > 1$, since both $R_{event}(\beta)$ and affects(β) are irreflexive.

Since the relation $R_{event}(\beta)$ is acyclic, there must be at least one index i such that $(\pi_i,\pi_{i+1}) \in$ affects(β) and $(\pi_i,\pi_{i+1}) \notin R_{event}(\beta)$. Let T and T' be the lowtransactions of π_i and π_{i+1} respectively. By the condition in the hypothesis, T is either an ancestor or a descendant of T'. We consider two cases.

1. T is an ancestor of T'.

   If the pair (π_{i-1},π_i) is in affects(β), then by the transitivity of the affects relation, $(\pi_{i-1},\pi_{i+1}) \in$ affects(β). On the other hand, if $(\pi_{i-1},\pi_i) \in R_{event}(\beta)$, then by Lemma 24, $(\pi_{i-1},\pi_{i+1}) \in R_{event}(\beta)$. In either situation, there is a shorter cycle in the relation $R_{event}(\beta) \cup$ affects(β), obtained by omitting π_i. This contradicts our assumption that the cycle chosen is as short as possible.

2. T is a descendant of T'.

   If the pair (π_{i+1},π_{i+2}) is in affects(β), then by the transitivity of the affects

relation, $(\pi_i,\pi_{i+2}) \in$ affects(β). On the other hand, if $(\pi_{i+1},\pi_{i+2}) \in R_{event}(\beta)$, then by Lemma 24, $(\pi_i,\pi_{i+2}) \in R_{event}(\beta)$. In either situation, there is a shorter cycle in the relation $R_{event}(\beta) \cup$ affects(β), obtained by omitting π_{i+1}. This contradicts the assumption that the cycle chosen is as short as possible.

In every case, we have found a contradiction; thus, the assumption that the relation $R_{event}(\beta) \cup$ affects(β) contains a cycle must be wrong.

6.3. The Serializability Theorem

We now present the main result. It says that a simple behavior β is serially correct for a non-orphan transaction name T provided that there is a suitable sibling order R for which a certain "view condition" holds for each object name X. The view condition says that the portion of β occurring at X that is visible to T, reordered according to R, is a behavior of the serial object S_X. In order to make all of this precise, suppose β is a finite simple behavior, T a transaction name, R a sibling order that is suitable for β and T, and X an object name. Let ξ be the sequence consisting of those operations occurring in β whose transaction components are accesses to X and that are visible to T in β, ordered according to R_{trans} on the transaction components. (Lemma 25 implies that this ordering is uniquely determined.) Define *view*(β,T,R,X) to be perform(ξ).

Thus, view(β,T,R,X) represents the portion of the behavior β occurring at X that is visible to T, reordered according to R. Stated in other words, this definition extracts from β exactly the REQUEST_COMMIT actions for accesses to X that are visible to T; it then reorders those REQUEST_COMMIT actions according to R, and then inserts an appropriate CREATE action just prior to each REQUEST_COMMIT action. The theorem uses a hypothesis that each view(β,T,R,X) is a behavior of the serial object S_X to conclude that β is serially correct for T.

Theorem 27: (Serializability Theorem) Let β be a finite simple behavior, T a transaction name such that T is not an orphan in β, and R a sibling order suitable for β and T. Suppose that for each object name X, view(β,T,R,X) $\in$ finbehs(S_X). Then β is serially correct for T.

The theorem has a straightforward corollary that applies to other systems besides simple systems.

Corollary 28: Let $\{B_i\}_{i\in I}$ be a strongly compatible set of automata and let $B = \Pi_{i\in I}B_i$. Suppose that all non-access transaction names T are in the index set I and that A_T and B_T are identical automata for all such T.

Let β be a finite behavior of B, T a transaction name that is not an orphan in β and R a sibling order suitable for serial(β) and T. Suppose that the following conditions hold.

1. serial(β) is a simple behavior.

2. For each object name X, view(serial(β),T,R,X) $\in$ finbehs(S_X).

Then β is serially correct for T.

We use the Serializability Theorem and its corollary later in this paper to reason about two locking algorithms, and in [AFLMW] to prove correctness of timestamp algorithms. The rest of this section contains a careful (and somewhat technical) proof of the Serializability Theorem.

The reader who is more interested in the applications of this theorem than in its proof may wish to go on to later sections without reading the rest of this section. Nothing in the rest of this section is needed for understanding the rest of the paper.

6.4. Proof of the Serializability Theorem

This subsection is devoted to a proof of the Serializability Theorem. In this subsection, several technical terms are defined, such as "ordered-visible" and "pictures". These definitions are not used elsewhere in the paper.

The general strategy is as follows. Given a finite simple behavior β, a non-orphan transaction T, and a suitable sibling order R, we must produce a serial behavior γ that looks the same as β to T, i.e. such that $\beta|T = \gamma|T$. The construction of γ is done in three steps. First, visible(β,T), the portion of β visible to T, is extracted from β. Second, this sequence is reordered according to R and affects(β). (There may be many ways of doing this.) The set of all acceptable reorderings is called *ordered-visible*(β,T). Third, we take a prefix γ of a sequence in ordered-visible(β,T) that includes all events of T. The set of all acceptable such prefixes is called *pictures*(β,T,R). We argue that γ is the required serial behavior by showing separately that its projections are behaviors of the transaction automata, of the serial object automata, and of the serial scheduler, and then applying Proposition 2.

6.4.1. Pictures

If β is a finite simple behavior, T a transaction name and R a suitable sibling order for β and T, then define *ordered-visible*(β,T,R) to be the set of reorderings of visible(β,T) that are consistent with affects(β) $\cup$ $R_{event}(\beta)$. Also, define *pictures*(β,T,R) to be the set of all sequences γ obtained as follows. If no actions π with transaction(π) = T appear in visible(β,T) then γ is the empty sequence. Otherwise, take a sequence δ in ordered-visible(β,T,R). Then γ is the prefix of δ ending with π, where π is the last event in δ such that hightransaction(π) is a descendant of T.

Lemma 29: Let β be a finite simple behavior, T a transaction name and R a suitable sibling order for β and T. Then ordered-visible(β,T,R) and pictures(β,T,R) are nonempty sets of sequences.

Proof: By the fact that R is suitable for β and T.

Lemma 30: Let β be a finite simple behavior, T a transaction name and R a sibling order that is suitable for β and T. Let $\gamma \in$ pictures(β,T,R). If ϕ and π are events of β, ϕ affects π in β and π is an event of γ, then ϕ is an event in γ, and ϕ precedes π in γ.

Proof: Since affects(β) is the transitive closure of the finite relation directly-affects(β), it suffices to prove the lemma in the case that ϕ directly-affects π in β. Since π is in visible(β,T), examination of the six cases of the definition of directly-affects(β) shows that ϕ is also in visible(β,T). By definition, γ is a prefix of a sequence δ in ordered-visible(β,T,R). Since δ is ordered consistently with affects(β), ϕ precedes π in δ. Therefore, ϕ is in γ.

6.4.2. Behavior of Transactions

In this subsection, we show that any sequence in pictures(β,T,R) projects to yield a finite behavior of each transaction automaton.

Lemma 31: Let β be a simple behavior, T a transaction name and R a sibling order

that is suitable for β and T. Suppose $\gamma \in$ pictures(β,T,R). Then $\gamma|T = \beta|T$, and $\gamma|T'$ is a prefix of $\beta|T'$ for all transaction names T'.

Proof: By the definition of pictures, using Lemma 18 and the fact that the directly-affects relation orders all events in β with the same transaction.

Lemma 32: Let β be a simple behavior, T a transaction name and R a sibling order that is suitable for β and T. Suppose $\gamma \in$ pictures(β,T,R). Then $\gamma|T'$ is a finite behavior of $A_{T'}$ for every non-access transaction name T'.

Proof: By Lemma 31 and Proposition 1.

6.4.3. Behavior of Serial Objects

Next we show that any sequence in pictures(β,T,R) projects to yield a finite behavior of each serial object automaton. We will use the view condition to show this; thus, we must begin by relating the definitions of "view" and "pictures".

Lemma 33: Let β be a finite simple behavior, T a transaction name and R a sibling order suitable for β and T. Let $\delta \in$ ordered-visible(β,T,R). Let X be an object name. Then one of the following two possibilities holds.

1. $\delta|X$ is identical to view(β,T,R,X).
2. T is an access to X and $\delta|X$ is the result of inserting a single CREATE(T) event somewhere in the sequence view(β,T,R,X).

Proof: The two constructions imply that $\delta|X$ and view(β,T,R,X) have identical subsequences of REQUEST_COMMIT actions. The sequence view(β,T,R,X)) contains exactly one CREATE(U) immediately preceding each REQUEST_COMMIT for U. Each such CREATE(U) also appears in $\delta|X$, by the preconditions for the simple database and the definition of visibility; moreover, the definition of ordered-visible implies that each such CREATE(U) also appears immediately preceding the corresponding REQUEST_COMMIT for U. Thus, the only possible difference between $\delta|X$ and view(β,T,R,X) is that $\delta|X$ might contain some extra CREATE(U) events, without matching REQUEST_COMMIT events for U.

Since δ is a reordering of a subsequence of visible(β,T), any such unmatched CREATE(U) event must have U visible to T in β. Since no REQUEST_COMMIT for U appears in $\delta|X$, none appears in visible(β,T) and hence none appears in β. Simple database preconditions imply that no COMMIT(U) appears in β. Therefore, it must be that U = T, and that T is an access to X.

Lemma 34: 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. Let $\gamma \in$ pictures(β,T,R). Let X be an object name. Then $\gamma|X$ is either a prefix of view(β,T,R,X) or else is a prefix of view(β,T,R,X) followed by a single CREATE(T) event.

Proof: By definition of pictures(β,T,R), γ is obtained as a prefix of a sequence $\delta \in$ ordered-visible(β,T,R). The previous lemma implies that $\delta|X$ and view(β,T,R,X) are identical except that an extra CREATE(T) event might appear in $\delta|X$, and this can only occur in case T is an access to X.

If $\delta|X$ contains no extra CREATE events not present in view(β,T,R,X), then it is immediate by the construction of γ as a prefix of δ that $\gamma|X$ is a prefix of view(β,T,R,X), as needed. So suppose that $\delta|X$ is the same as view(β,T,R,X) except

that $\delta|X$ contains an extra CREATE(T) event. Then the definition of pictures implies that $\gamma|X$ is the prefix of $\delta|X$ ending with the CREATE(T) event. Then $\gamma|X$ is a prefix of view(β,T,R,X) followed by a single CREATE(T) event.

Lemma 35: Let β be a simple behavior, T a transaction name, R a sibling order that is suitable for β and T, and X an object name. Suppose that view(β,T,R,X) is a finite behavior of S_X. Suppose $\gamma \in$ pictures(β,T,R). Then $\gamma|X$ is a finite behavior of S_X.

Proof: By Lemma 34 and the fact that inputs to S_X, as with any I/O automaton, are always enabled.

6.4.4. Behavior of the Serial Scheduler

Next, we show that any sequence in pictures(β,T,R) is a behavior of the serial scheduler.

Lemma 36: 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 that is suitable for β and T. Let $\gamma \in$ pictures(β,T,R). Then γ is a finite behavior of the serial scheduler.

Proof: By definition of pictures(β,T,R), γ is obtained as a prefix of a sequence $\delta \in$ ordered-visible(β,T,R). That is, if no actions π with transaction(π) = T appear in visible(β,T) then γ is empty. Otherwise, γ is the prefix of δ ending with the last event in δ that has hightransaction a descendant of T.

The proof is by induction on prefixes of γ, with a trivial basis. Let $\gamma'\pi$ be a prefix of γ with π a single event, and assume that γ' is a behavior of the serial scheduler. If π is an input action of the serial scheduler, then the fact that inputs are always enabled implies that γ is a behavior of the serial scheduler. So assume that π is an output action of the serial scheduler. Let s' be the state of the serial scheduler after γ'. We must show that π is enabled in the serial scheduler automaton in state s'.

1. π is CREATE(T'). We show that T' $\in$ s'.create_requested - s'.created - s'.aborted and that siblings(T') $\cap$ s'.created $\subseteq$ s'.completed.

   By the preconditions of the simple database and Lemma 12, a REQUEST_CREATE(T') event ϕ precedes π in β. Then $(\phi,\pi) \in$ affects(β), so Lemma 30 implies that ϕ is in γ'. Thus T' $\in$ s'.create_requested.

   Since only one CREATE(T') occurs in β, no CREATE(T') occurs in γ', so by Lemma 8, T' $\notin$ s'.created.

   Since by Lemma 17, T' is not an orphan in β, no ABORT(T') occurs in β. Thus, no ABORT(T') occurs in γ', so by Lemma 8, T' $\notin$ s'.aborted.

   Suppose T'' is a sibling of T' that is in s'.created. Then CREATE(T'') occurs in γ', by Lemma 12. Since the order of events in γ is consistent with $R_{event}(\beta)$, (T',T'') $\notin R_{trans}$. Since R_{trans} is suitable for β and T, (T'',T') $\in R_{trans}$. If T is a descendant of T'', then T and T' are incomparable and so (T,T') $\in R_{trans}$. Since δ is ordered consistently with $R_{event}(\beta)$, π follows all events ϕ with hightransaction(ϕ) a descendant of T, in δ. But then the definition of pictures would exclude π from γ, a contradiction. Therefore, T is not a descendant of T''. Since T'' is visible to T in β, a COMMIT(T'') event occurs in β. This COMMIT(T'') is in visible(β,T) and is ordered before π by $R_{event}(\beta)$. Thus, COMMIT(T'') precedes π in δ, and so COMMIT(T'') occurs in γ'. Hence, T'' $\in$ s'.completed.

2. π is COMMIT(T').

We show that (T',v) $\in$ s'.commit_requested for some v, and that T' $\notin$ s'.completed.

By the preconditions of the simple database, there is a value v such that a REQUEST_COMMIT(T',v) event ϕ appears in β. Then $(\phi,\pi) \in$ affects(β), so Lemma 30 implies that ϕ is in γ'. Thus (T',v) $\in$ s'.commit_requested.

By Lemma 13, there is only one completion event for T' in β and hence only one in γ. Hence, T' $\notin$ s'.completed.

3. π is ABORT(T').

We must show that T' $\in$ s'.create_requested - s'.completed - s'.created and siblings(T') $\cap$ s'.created $\subseteq$ s'.completed.

By the preconditions of the simple database, a REQUEST_CREATE(T') event ϕ appears in β. Then $(\phi,\pi) \in$ affects(β), so Lemma 30 implies that ϕ is in γ'. Thus, T' $\in$ s'.create_requested.

Since by Lemma 13 there is at most one completion event in β, there can be no completion event in γ'. Thus, T' $\notin$ s'.completed.

Also T' is an orphan in β, so by Lemma 17, T' is not visible to T in β. Thus CREATE(T') does not occur in visible(β,T) and so also CREATE(T') does not occur in γ. Thus, T' $\notin$ s'.created.

Suppose T'' is a sibling of T' that is in s'.created, so that CREATE(T'') occurs in γ'. Since the order of events in γ is consistent with $R_{event}(\beta)$, (T',T'') $\notin R_{trans}$. Since R_{trans} is suitable for β and T, (T'',T') $\in R_{trans}$. If T is a descendant of T'', then T and T' are incomparable and so (T,T') $\in R_{trans}$. Since δ is ordered consistently with $R_{event}(\beta)$, π follows all events ϕ with hightransaction(ϕ) a descendant of T, in δ. But then the definition of pictures would exclude π from γ, a contradiction. Therefore, T is not a descendant of T''. Since T'' is visible to T in β, a COMMIT(T'') event occurs in β. This COMMIT(T'') is in visible(β,T) and is ordered before π by $R_{event}(\beta)$. Thus, COMMIT(T'') precedes π in δ, and so COMMIT(T'') occurs in γ'. Hence, T'' $\in$ s'.completed.

4. π is a REPORT_COMMIT or REPORT_ABORT event for T'.

By the preconditions of the simple database and Lemma 12, a COMMIT or ABORT event ϕ appears in β. Then $(\phi,\pi) \in$ affects(β), so Lemma 30 implies that ϕ is in γ'. Also, by Lemma 13 there is at most one report event in β, so there can be no report event in γ'. Thus, T' $\notin$ s'.reported.

Thus, π is enabled in the serial scheduler in state s'.

6.4.5. Proof of the Main Result

We can now tie the pieces together to prove Theorem 27, the Serializability Theorem.

Proof: Let $\gamma \in$ pictures(β,T,R). (Lemma 29 implies that this set is nonempty.) Lemma 32 shows that $\gamma|T'$ is a finite behavior of $A_{T'}$, for all non-access transaction

names T'. Lemma 35 shows that $\gamma|X$ is a finite behavior of S_X, for all object names X. Lemma 36 implies that γ is a finite behavior of the serial scheduler. Proposition 2 implies that γ is a finite serial behavior. The definition of pictures implies that $\gamma|T = \beta|T$.

It is easy to see that the serial behavior γ constructed to show serial correctness for T_0 also has the property that $\gamma|T = \beta|T$ for all T visible to T_0 in β. Thus, if the view condition holds for a suitable sibling order for T_0, then there exists a single serial schedule that looks like β to all the transactions that commit to the top level.

7. Dynamic Atomicity

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

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

Finally, we define another condition called "local dynamic atomicity"; this is a convenient sufficient condition for proving dynamic atomicity.

7.1. Completion Order

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

1. There are completion events for both T and T' in β, and a completion event for T precedes a completion event for T'.

2. There is a completion event for T in β, but there is no completion event for T' in β.

The following is easy to see.

Lemma 37: Let β be a simple behavior. Then completion(β) is a sibling order.

The next few lemmas show that the completion order is suitable.

Lemma 38: Let β be a simple behavior and let R = completion(β). Let π and π' be distinct events in β with lowtransactions T and T' respectively. If T is neither an ancestor nor a descendant of T', and $(\pi,\pi') \in$ affects(β), then $(\pi,\pi') \in R_{event}(\beta)$.

Proof: Since T is neither an ancestor nor a descendant of T', there are siblings U and U' such that T is a descendant of U and T' is a descendant of U'. Since π affects π' in β, by Lemmas 21 and 22, there must be events ϕ and ϕ' in β such that ϕ is a completion

event for U, ϕ' is REQUEST_CREATE(U'), either $\pi = \phi$ or else $(\pi,\phi) \in$ affects(β), and (ϕ,ϕ') and (ϕ',π') are both in affects(β). Thus, ϕ occurs before ϕ' in β.

The simple database preconditions and transaction well-formedness imply that any completion event for U' in β must occur after the unique REQUEST_CREATE(U') event. Thus β contains a completion event ϕ for U, which precedes ϕ', which in turn precedes any completion event for U'. Thus $(U,U') \in R$ = completion(β), and therefore $(\pi,\pi') \in R_{event}(\beta)$.

Lemma 39: Let β be a simple behavior and let R = completion(β). Then $R_{event}(\beta)$ and affects(β) are consistent partial orders on the events of β.

Proof: Immediate by Lemmas 38 and 26.

Lemma 40: Let β be a simple behavior and T a transaction name. If T' and T'' are siblings that are lowtransactions of actions in visible(β,T) then either (T',T'') or (T'',T') $\in$ completion(β).

Proof: Since T' and T'' are distinct siblings, T is not a descendant of both T' and T''. Without loss of generality, we will assume that T is not a descendant of T'. Note that therefore the least common ancestor of T and T' must be an ancestor of parent(T'). There is an event π in visible(β,T) such that lowtransaction(π) = T'. Thus either π is a completion event for T' or hightransaction(π) must be T'. In the case where hightransaction(π) = T', we must have that T' is visible to T in β, and thus (since T' is not an ancestor of T) that β contains a COMMIT(T') event. Thus in either case β contains a completion event for T', and so completion(β) orders T' and T''.

Now we can conclude that the completion order is suitable.

Lemma 41: Let β be a finite simple behavior and T a transaction name. Then completion(β) is suitable for β and T.

Proof: By Lemmas 40 and 39.

7.2. Generic Systems

In this subsection, we give the system decomposition appropriate for describing locking algorithms. We will formulate such algorithms as "generic systems", which are composed of transaction automata, "generic object automata" and a "generic controller". The general structure of the system is the same as that given in Figure 1, for serial systems.

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

7.2.1. Generic Object Automata

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

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

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

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

As with previous well-formedness properties, we have an alternative form of the definition.

Lemma 42: Let $\beta\pi$ be a finite sequence of actions in the external signature of a generic object for object name X, where π is a single event. Then $\beta\pi$ is generic object well-formed for X exactly if β is generic object well-formed for X and the following conditions hold.

1. If π is CREATE(T), then
   a. there is no CREATE event for T in β'.
2. If π is INFORM_COMMIT_AT(X)OF(T), then
   a. if T is an access to X, then there is a REQUEST_COMMIT for T in β', and
   b. there is no INFORM_ABORT_AT(X)OF(T) in β'.
3. If π is INFORM_ABORT_AT(X)OF(T), then
   a. there is no INFORM_COMMIT_AT(X)OF(T) in β'.
4. If π is REQUEST_COMMIT(T,v), then
   a. there is a CREATE(T) in β', and
   b. there is no REQUEST_COMMIT for T in β'.

7.2.2. Generic Controller

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

The generic controller is a very nondeterministic automaton. It may delay passing requests or reports or making decisions for arbitrary lengths of time, and may decide at any time to abort a transaction whose creation has been requested (but that has not yet completed). Each specific implementation of a locking algorithm will make particular choices from among the many nondeterministic possibilities. For instance, Moss [21] devotes considerable effort to describing a particular distributed implementation of the controller that copes with node and communication failures yet still commits a subtransaction whenever possible. Our results apply *a fortiori* to all implementations of the generic controller obtained by restricting the nondeterminism.

The generic controller has the following action signature.

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

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

Each state s of the generic controller consists of six sets: s.create_requested, s.created, s.commit_requested, s.committed, s.aborted and s.reported. The set s.commit_requested is a set of (transaction,value) pairs, and the others are sets of transactions. All are empty in the start state except for create_requested, which is $\{T_0\}$. Define s.completed = s.committed $\cup$ s.aborted. The transition relation is as follows.

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

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

CREATE(T)
Precondition:
T ∈ s'.create_requested - s'.created
Effect:
s.created = s'.created ∪ {T}

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

ABORT(T), $T \neq T_0$
Precondition:
T ∈ s'.create-requested - s'.compieted
Effect:
s.aborted = s'.aborted ∪ {T}

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition:
T ∈ s'.committed
(T,v) ∈ s'.commit_requested
T ∉ s'.reported
Effect:
T ∈ s.reported

REPORT_ABORT(T), $T \neq T_0$
Precondition:
T ∈ s'.aborted
T ∉ s'.reported
Effect:
T ∈ s.reported

INFORM_COMMIT_AT(X)OF(T), $T \neq T_0$
Precondition:
T ∈ s'.committed

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Precondition:
T ∈ s'.aborted

Lemma 43: Let β be a finite schedule of the generic controller, and let s be a state such that β can leave the generic controller in state s. Then the following conditions are true.

1. T is in s.create_requested exactly if $T = T_0$ or β contains a REQUEST_CREATE(T) event.
2. T is in s.created exactly if β contains a CREATE(T) event.
3. (T,v) is in s.commit_requested exactly if β contains a

REQUEST_COMMIT(T,v) event.

4. T is in s.committed exactly if β contains a COMMIT(T) event.
5. T is in s.aborted exactly if β contains an ABORT(T) event.
6. T is in s.reported exactly if β contains a report event for T.
7. s.committed $\cap$ s.aborted = $\emptyset$.
8. s.reported $\subseteq$ s.committed $\cup$ s.aborted.

Lemma 44: Let β be a schedule of the generic controller. Then all of the following hold:

1. If a CREATE(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
2. At most one CREATE(T) event appears in β for each transaction T.
3. If a COMMIT(T) event appears in β, then a REQUEST-COMMIT(T,v) event precedes it in β for some return value v.
4. If an ABORT(T) event appears in β, then a REQUEST-CREATE(T) event precedes it in β.
5. At most one completion event appears in β for each transaction.
6. At most one report event appears in β for each transaction.
7. If a REPORT-COMMIT(T,v) event appears in β, then a COMMIT(T) event and a REQUEST_COMMIT(T,v) event precede it in β.
8. If a REPORT-ABORT(T) event appears in β, then an ABORT(T) event precedes it in β.

7.2.3. Generic Systems

A *generic system* of a given system type is the composition of a strongly compatible set of automata indexed by the union of the set of non-access transaction names, the set of object names and the singleton set {GC} (for "generic controller"). Associated with each non-access transaction name T is a transaction automaton A_T for T, the same automaton as in the serial system. Associated with each object name X is a generic object automaton G_X for X. Finally, associated with the name GC is the generic controller automaton for the system type.

The external actions of a generic system are called *generic actions*, and the executions, schedules and behaviors of a generic system are called *generic executions*, *generic schedules* and *generic behaviors*, respectively. The following proposition says that generic behaviors have the appropriate well-formedness properties. Its proof is analogous to that of the similar result for serial behaviors.

Proposition 45: If β is a generic behavior, then the following conditions hold.

1. For every transaction name T, $\beta|T$ is transaction well-formed for T.
2. For every object name X, $\beta|G_X$ is generic object well-formed for X.

The following result says that if the INFORM events are removed from any generic behavior, the result is a simple behavior.

Proposition 46: If β is a generic behavior then serial(β) is a simple behavior.

Proof: By a straightforward induction on the length of β.[16]

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

Proposition 47: Let β be a finite generic behavior, T a transaction name that is not an orphan in β and R = completion(β). Suppose that for each object name X, view(serial(β),T,R,X) ∈ finbehs(S_X). Then β is serially correct for T.

Proof: Immediate from Corollary 28, using Lemma 41, Proposition 46, and the observation that completion(β) = completion(serial(β)).

7.3. Dynamic Atomicity

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

This proof structure can be used to yield much stronger results than just the correctness of the locking algorithms in this paper. As long as each object is dynamic atomic, the whole system will guarantee that any finite behavior is serially correct for all non-orphan transaction names. Thus, we are free to use an arbitrary implementation for each object, independent of the choice of implementation for each other object, as long as dynamic atomicity is satisfied. For example, a simple algorithm such as Moss's can be used for most objects, while a more sophisticated algorithm permitting extra concurrency by using type-specific information can be used for objects that are "hot spots". (That is, objects that are very frequently accessed.) The idea of a condition on objects that guarantees serial correctness was introduced by Weihl [27] for systems without transaction nesting.

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

Theorem 48: (Dynamic Atomicity Theorem) Let $\mathcal{S}$ be a generic system in which all generic objects are dynamic atomic. Let β be a finite behavior of $\mathcal{S}$. Then β is serially correct for every non-orphan transaction name.

Proof: Immediate from Proposition 47 and the definition of dynamic atomicity.

---

[16] An alternative proof can be formulated in terms of the notion of implementation, using a possibilities mapping.

7.4. Local Dynamic Atomicity

In the previous subsection, we showed that to prove that a generic system guarantees serial correctness for non-orphan transactions it is enough to check that each generic object automaton is dynamic atomic. In this subsection, we define another property of generic object automata called "local dynamic atomicity", which is a convenient sufficient condition for showing dynamic atomicity. For each generic object automaton G, dynamic atomicity is a local condition in that it only depends on G. However, the form in which the condition is stated may be difficult to check directly: one must be able to verify a condition involving view(serial(β),T,completion(β),X) for all finite behaviors β of all generic systems containing G. Local dynamic atomicity is defined more directly in terms of the behaviors of G.

First we introduce some terms to describe information about the the status of transactions that is deducible from the behavior of a particular generic object. Let G be a generic object automaton for X, β a sequence of external actions of G, and T and T' transaction names. Then T is *locally visible* to T' in β if β contains an INFORM_COMMIT_AT(X)OF(U) event for every U in ancestors(T) - ancestors(T'). Also, T is a *local orphan* in β if an INFORM_ABORT_AT(X)OF(U) event occurs in β for some ancestor U of T. The following are obvious facts about local visibility and local orphans.

Lemma 49: Let G be a generic object automaton for X. Let β be a sequence of external actions of G, and let T, T' and T'' be transaction names. If T is locally visible to T' in β, and T' is locally visible to T'' in β, then T is locally visible to T'' in β.

Lemma 50: Let G be a generic object automaton for X. Let β be a generic object well-formed sequence of external actions of G, and let T and T' be transaction names. If T is locally visible to T' in β, and T' is not a local orphan in β, then T is not a local orphan in β.

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

Lemma 51: Let β be a behavior of a generic system in which generic object automaton G is associated with X. If T is locally visible to T'in β|G then T is visible to T' in β. Similarly, if T is a local orphan in β|G then T is an orphan in β.

Proof: These are immediate consequences of the generic controller preconditions, which imply that any INFORM_ABORT_AT(X)OF(T) event in β must be preceded by an ABORT(T) event and that any INFORM_COMMIT_AT(X)OF(T) is preceded by COMMIT(T).

Next, we define a relation on accesses to the generic object G to describe some information about the completion order that is deducible from the behavior of a particular generic object. Given a sequence β of external actions of a generic object automaton for X, we define a binary relation local-completion(β) on accesses to X. Namely, (U,U') ∈ local-completion(β) exactly if U ≠ U', β contains REQUEST_COMMIT events for both U and U', and U is locally visible to U' in β', where β' is the longest prefix of β not containing the given REQUEST_COMMIT event for U'.

Lemma 52: If β is a generic object well-formed sequence of external actions of a generic object automaton for X, then local-completion(β) is an irreflexive partial order on accesses to X.

Proof: We must show that local-completion(β) is irreflexive, antisymmetric and

transitive. Irreflexivity follows immediately from the definition.

Suppose that (T,T') and (T',T) are both in local-completion(β). Then β contains a REQUEST_COMMIT event for each of T and T', and generic object well-formedness implies that there is only one of each. Since (T,T') ∈ local-completion(β), T is locally visible to T' in the longest prefix β' of β not containing the REQUEST_COMMIT for T'. Therefore, an INFORM_COMMIT for T occurs in β', and generic object well-formedness implies that the REQUEST_COMMIT for T precedes the REQUEST_COMMIT for T' in β. But the same reasoning implies that the REQUEST_COMMIT for T' precedes the REQUEST_COMMIT for T in β, a contradiction. Therefore, local-completion(β) is antisymmetric.

Now suppose (T,T') and (T',T'') are both in local-completion(β). Let β' and β'' be the longest prefixes of β not containing a REQUEST_COMMIT for T' and not containing a REQUEST_COMMIT for T', respectively. As in the argument above, the REQUEST_COMMIT for T' must precede the REQUEST_COMMIT for T'' in β, so β' is a prefix of β''. Since T is locally visible to T' in β', T is locally visible to T' in β'', and since T' is locally visible to T'' in β'', Lemma 50 implies that T is locally visible to T'' in β''. Thus (T,T'') ∈ local-completion(β).

The relationship between the local-completion order and the true completion order in a generic system is as follows.

Lemma 53: Let β be a behavior of a generic system in which generic object automaton G is associated with X. Let T and T' be accesses to X. If (T,T') ∈ local-completion(β|G), and T' is not an orphan in β, then (T,T') ∈ R_{trans}, where R = completion(β).

Proof: By definition of local-completion(β), β|G contains a REQUEST_COMMIT event for T', and T is locally visible to T' in β'|G, where β' is the longest prefix of β not containing the REQUEST_COMMIT for T'. Lemma 51 implies that T is visible to T' in β'.

Since β is well-formed, it contains at most one REQUEST_COMMIT event for T', and so β' does not contain a REQUEST_COMMIT event for T'. By the controller preconditions, and Lemma 44, β' does not contain a COMMIT(T') event. Since β|G is generic object well-formed, β' contains a CREATE(T') event. Since T' is not an orphan in β, β' does not contain an ABORT(T') event. Therefore, T' is live in β'.

Let U and U' denote the siblings such that T is a descendant of U, and T' is a descendant of U'. Since T is visible to T' in β', β' contains a COMMIT(U) event. By Lemmas 46 and 16, U' must be live in β'. Since β' contains a return for U, and no return for U', it follows that (U,U') ∈ R. Therefore (T,T') ∈ R_{trans}.

Now we give a definition to describe how to reorder the external actions of a generic object automaton according to a given local-completion order. Suppose β is a generic object well-formed sequence of external actions of a generic object automaton for X and T is a transaction name. Let *local-pictures*(β,T) be the set of sequences defined as follows. Let Z be the set of operations occurring in β whose transactions are locally visible to T in β. Then the elements of local-pictures(β,T) are the sequences of the form perform(ξ), where ξ is a total ordering of Z in an order consistent with the partial order local_completion(β) on the transaction components. The following is straightforward from the definitions.

Lemma 54: If β is a generic object well-formed sequence of external actions of a generic object automaton for X and T is a transaction name, then every element of local-pictures(β,T) is serial object well-formed.

We are finally ready to define "local dynamic atomicity". We say that generic object automaton G for object name X is *locally dynamic atomic* if whenever β is a finite generic object well-formed behavior of G and T is a transaction name that is not a local orphan in β, then local-pictures(β,T) $\subseteq$ finbehs(S_X). That is, the result of reordering a behavior of G according to the given local-completion order is a finite behavior of the corresponding serial object automaton. The main result of this subsection says that local dynamic atomicity is a sufficient condition for dynamic atomicity.

Theorem 55: If G is a generic object automaton for object name X that is locally dynamic atomic then G is dynamic atomic.

Proof: Let $\mathcal{S}$ be a generic system in which G is associated with X. Let β be a finite behavior of $\mathcal{S}$, R = completion(β) and T a transaction name that is not an orphan in β. We must prove that view(simple(β),T,R,X) $\in$ finbehs(S_X). By definition, view(simple(β),T,R,X) = perform(ξ), where ξ is the sequence of operations occurring in β whose transactions are visible to T in β, arranged in the order given by R_{trans} on the transaction component.

Let γ be a finite sequence of actions consisting of exactly one INFORM_COMMIT_AT(X)OF(U) for each COMMIT(U) that occurs in β. Then $\beta\gamma$ is a behavior of the system $\mathcal{S}$, since each action in γ is an enabled output action of the generic controller, by Lemma 43. Then $\beta\gamma$|G is a behavior of G, and Lemma 45 implies that it is generic object well-formed.

Since INFORM_COMMIT_AT(X)OF(U) occurs in $\beta\gamma$|G if and only if COMMIT(U) occurs in β, an access T' to X is visible to T in β if and only if it is locally visible to T in $\beta\gamma$|G. Therefore, the same operations occur in view(simple(β),T,R,X) and in any sequence in local-pictures($\beta\gamma$|G,T). To show that view(simple(β),T,R,X) $\in$ local-pictures($\beta\gamma$|G,T), we must show that they can appear in the same order.

If T' is any access that is locally visible to T in $\beta\gamma$|G, then T' is visible to T in β, so Lemma 17 implies that T' is not an orphan in β, and hence not an orphan in $\beta\gamma$. Also, note that completion($\beta\gamma$) = completion(β) = R. Then Lemma 53 implies that if accesses that are locally visible to T in $\beta\gamma$|G are ordered by local-completion($\beta\gamma$|G), they are also ordered in the same way by R_{trans}.

Thus, the sequence ξ can be obtained by taking those operations (T',v') such that REQUEST_COMMIT(T',v') occurs in $\beta\gamma$|G and T' is locally visible to T in $\beta\gamma$|G, and arranging them in an order that is consistent with local-completion($\beta\gamma$|G) on the transaction component. Thus, perform(ξ) is an element of local-pictures($\beta\gamma$|G,T). Since G is locally dynamic atomic, perform(ξ) is a finite behavior of S_X, as required.

8. Restricted Types of Serial Objects

The correctness of the two algorithms in this paper depends on semantic information about the types of serial object automata used in the underlying serial system. For example, Moss's algorithm provides special treatment for "read accesses", i.e., accesses that do not modify the

state of the object. Also, our general commutativity-based locking algorithm uses information about commutativity of certain operations in order to determine the orders in which these operations are permitted to occur. In this section, we provide the appropriate definitions for these concepts.

We first define the important concept of "equieffectiveness" of two sequences of external actions of a serial object automaton. Roughly speaking, two sequences are "equieffective" if they can leave the automaton in states that are indistinguishable to the outside world. We then define the notion of "commutativity" required for our algorithm. Finally, we define "read accesses"; that is, we state the properties of read accesses that are required for the correctness of Moss's algorithm.

8.1. Equieffectiveness

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

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

The following proposition says that extensions of equieffective sequences are also equieffective.

Proposition 56: Let X be an object name. Let β and β' be equieffective sequences of actions in $ext(S_X)$. Let γ be a finite sequence of actions in $ext(S_X)$. Then $\beta\gamma$ is equieffective to $\beta'\gamma$.

Equieffectiveness is not an equivalence relation, but we do have a restricted transitivity result.

Lemma 57: Let X be an object name, and let ξ, η and ζ be three serial object well-formed finite sequences of operations of X, such that every operation in η appears in either ξ or ζ. If $perform(\xi)$ is equieffective to $perform(\eta)$, and $perform(\eta)$ is equieffective to $perform(\zeta)$, then $perform(\xi)$ is equieffective to $perform(\zeta)$.

Proof: Suppose $perform(\xi)$ and $perform(\eta)$ are equieffective, and that $perform(\eta)$ and $perform(\zeta)$ are equieffective. Let γ be a sequence of external actions of S_X such that $perform(\xi)\gamma$ and $perform(\zeta)\gamma$ are serial object well-formed, and suppose that $perform(\xi)\gamma$ is a behavior of S_X. We show that $perform(\zeta)\gamma$ is a behavior of S_X.

By the definition of serial object well-formedness, γ must be either of the form $perform(\tau)$ or $perform(\tau)CREATE(T)$, where the first components of all the operations

in τ (and T as well, if appropriate) are distinct from the first components of all the operations in ξ and ζ. By the condition on η, the first components of all the operations in τ (and T as well, if appropriate) are distinct from the first components of the operations in η. Thus, perform(η)β is serial object well-formed. The definition of equieffectiveness than implies that perform(η)γ is a behavior of S_X, and therefore that perform(ζ)γ is a behavior of S_X, as needed.

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

8.2. Commutativity

We now define an appropriate notion of commutativity for operations of a particular serial object automaton.[17] Namely, we say that operations (T,v) and (T',v') *commute*, where T and T' are accesses to X, if for any sequence of operations ξ such that both perform(ξ(T,v)) and perform(ξ(T',v')) are serial object well-formed behaviors of S_X, then perform(ξ(T,v)(T',v')) and perform(ξ(T',v')(T,v)) are equieffective serial object well-formed behaviors of S_X.

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

- balance?: The return value for this access gives the current balance.
- deposit_$a: This increases the balance by $a. The only return value is "OK".
- withdraw_$b: This reduces the balance by $b if the result will not be negative. In this case the return value is "OK". If the result of withdrawing would be to cause an overdraft, then the balance is left unchanged, and the return value is "FAIL".

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

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

occur. Finally, the balance left after each of β and β' is \$(x+b+b'), where \$x is the balance after perform(ξ), so β and β' are equieffective.

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

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

A consequence of the definition of commutativity is the following extension to sequences of operations.

> **Proposition 58:** Suppose that ζ and ζ' are finite sequences of operations of X such that each operation in ζ commutes with each operation in ζ'. If ξ is a finite sequence of operations of S_X such that perform($\xi\zeta$) and perform($\xi\zeta'$) are serial object well-formed behaviors of S_X, then perform($\xi\zeta\zeta'$) and perform($\xi\zeta'\zeta$) are equieffective serial object well-formed behaviors of S_X.

8.3. Transparent Operations

We now define the essential property that we will require of any read access. We say that an operation (T,v) at X is *transparent* if for any finite sequence β of external actions of S_X such that βperform(T,v) is a serial object well-formed behavior of S_X, βperform(T,v) and β are equieffective behaviors of S_X. Thus, a transparent operation does not affect the later behavior of the object automaton. The following simple extension shows that any subsequence consisting of transparent operations can be removed from a behavior, resulting in a behavior equieffective to the original one.

> **Proposition 59:** Let η be a finite serial object well-formed sequence of operations of X such that perform(η) is a behavior of S_X, and let ξ be a subsequence of η such that every operation in η-ξ is transparent. Then perform(η) and perform(ξ) are equieffective serial object well-formed behaviors of S_X.

It is easy to see that transparent operations commute.

> **Proposition 60:** Let (T,v) and (T',v') be transparent operations of X such that $T \neq T'$. Then (T,v) commutes with (T',v').
>
> **Proof:** Suppose ξ is a finite sequence of operations of X such that perform(ξ(T,v)) and perform(ξ(T',v')) are serial object well-formed behaviors of S_X. Then no operation in ξ has T or T' as first component, and all the operations in ξ have distinct first components. Therefore perform(ξ(T,v)(T',v')) and perform(ξ(T',v')(T,v)) are serial object well-formed sequences of external actions of S_X. Now perform(ξ(T,v))

and perform(ξ) are equieffective, since (T,v) is transparent. Since perform(ξ)perform(T',v') is a behavior of S_X, the definition of equieffectiveness implies that perform(ξ(T,v))perform(T',v') = perform(ξ(T,v)(T',v')) is also a behavior of S_X. Similarly, the fact that (T',v') is transparent implies that perform(ξ(T',v')(T,v)) is a behavior of S_X. By Proposition 59, each of perform(ξ(T,v)(T',v')) and perform(ξ(T',v')(T,v)) is equieffective to perform(ξ). Lemma 57 now shows that they are equieffective to each other, as required.

9. General Commutativity-Based Locking

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

9.1. Locking Objects

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

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

The transition relation of L_X is given by all triples (s',π,s) satisfying the following pre- and postconditions, given separately for each π. As before, any component of s not mentioned in the postconditions is the same in s as in s'.

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

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

INFORM_ABORT_AT(X)OF(T), T $\neq$ T_0

Effect:
s.intentions(U) = λ, U ∈ descendants(T)
s.intentions(U) = s'.intentions(U), U ∉ descendants(T)

REQUEST_COMMIT(T,v), T an access to X
Precondition:
T ∈ s'.created - s'.commit-requested
(T,v) commutes with every (T',v') in s'.intentions(U),
where U ∉ ancestors(T)
perform(total(s',T)(T,v)) ∈ finbehs(S_X)
Effect:
s.commit-requested = s'.commit-requested ∪ {T}
s.intentions(T) = s'.intentions(T)(T,v)
s.intentions(U) = s'.intentions(U) for U ≠ T

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

The following lemma says that the ordering of operations in the "total" sequences does not change during execution of L_X.

Lemma 61: Let $\beta_1\beta_2$ be a finite generic object well-formed schedule of L_X, such that β_1 can leave L_X in state s' and (s',β_2,s) is an extended step of L_X. Let T_1, T_2 and U and V be transaction names. Suppose (T_1,v_1) precedes (T_2,v_2) in total(s',U) and (T_2,v_2) occurs in total(s,V). Then (T_1,v_1) occurs in total(s,V) and precedes (T_2,v_2) in total(s,V).

The locking object L_X is quite nondeterministic; implementations[18] of L_X can be designed that restrict the nondeterminism in various ways, and correctness of such algorithms follows immediately from the correctness of L_X, once the implementation relationship has been proved, for example by using a possibilities mapping.

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

---

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

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

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

Another aspect of our algorithm that one might wish to change in an implementation is the complicated data structure maintaining the "intentions", and the corresponding need to replay all the operations recorded there when determining the response to an access. In the next section, we will consider an algorithm that is able to summarize all these lists of operations in a stack of versions of the serial object, at the cost of reducing available concurrency by using a conflict relation in which all updates exclude one another.

9.2. Correctness Proof

In this subsection, we prove several lemmas about L_X, leading to the theorem that L_X is dynamic atomic.

We first introduce some terms to describe the information L_X uses about visibility and about the completion order of transactions. If β is a sequence of actions of L_X, T is an access to X, and T' is an ancestor of T, we say that T is *lock-visible* 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). We also define a binary relation *lock-completion*(β) on accesses to X, where (U,U') $\in$ lock-completion(β) exactly if U $\neq$ U', β contains REQUEST_COMMIT events for both U and U', and U is lock-visible to U' in β', where β' is the longest prefix of β not containing the given REQUEST_COMMIT event for U'. The following simple lemma relates lock-visibility and the lock-completion order to local visibility and the local completion order. It follows immediately from the definitions.

Lemma 62: Let β be a generic object well-formed sequence of actions of L_X. Then

lock-completion(β) is an irreflexive partial order.

Lemma 63: Let β be a sequence of actions of L_X and T and T' transaction names. If T is lock-visible to T' in β then T is locally visible to T' in β. Also lock-completion(β) is a subrelation of local-completion(β).

The following lemma, which can be proved by a straightforward induction, shows which locks are held by a transaction after a schedule of L_X.

Lemma 64: Let β be a finite generic object well-formed schedule of L_X. Suppose that β can leave L_X in state s.

1. 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. Then (T,v) is a member of s.intentions(T').
2. If (T,v) is an element of s.intentions(T') then T is a descendant of T', REQUEST_COMMIT(T,v) occurs in β, and T' is the highest ancestor of T to which T is lock-visible in β.

We now give the key lemma, which shows that certain sequences of actions, extracted from a generic object well-formed behavior of L_X, are serial object well-formed behaviors of S_X. The second conclusion, that certain such sequences are equieffective, is needed to carry out the induction step of the proof of this lemma.

It is helpful to have an auxiliary definition. Suppose β is a generic object well-formed finite behavior of L_X. Then a set Z of operations of X is said to be *allowable* for β provided that for each operation (T,v) that occurs in Z, the following conditions hold.

1. (T,v) occurs in β.
2. T is not a local orphan in β.
3. If (T',v') is an operation that occurs in β such that (T',T) ∈ lock-completion(β), then (T',v') ∈ Z.

Lemma 65: Let β be a generic object well-formed finite behavior of L_X and let Z be an allowable set of operations for β. Let R = lock-completion(β).

1. If ξ is a total ordering of Z that is consistent with R on the transaction components, then perform(ξ) ∈ finbehs(S_X).
2. If ξ and η are both total orderings of Z such that each is consistent with R on the transaction component, then perform(ξ) and perform(η) are equieffective.

Proof: We use induction on the size of the set Z. The basis, when Z is empty, is trivial. So let $k \geq 1$ and suppose that Z contains k operations and the lemma holds for all allowable sets of k - 1 operations. Let ξ be a total ordering of Z that is consistent with R on the transaction component. Let (T,v) be the last operation in ξ, and let Z' = Z - {(T,v)}. Let ξ' be the sequence of operations such that ξ = ξ'(T,v). Then Z' is an allowable set of k - 1 operations, since Z is, and there is no operation (T',v') in Z such that (T,T') ∈ R. Also, ξ' is a total ordering of Z' consistent with R.

Let β' be the longest prefix of β not containing REQUEST_COMMIT(T,v), and let s' be the (unique) state in which β' can leave L_X. Let ζ_1 = total(s',T), and let ζ_2 be some total ordering that is consistent with R, of the operations in Z' - ζ_1. Lemma 64,

applied to β', implies that the operations in ζ_1 are exactly those (T',v') that occur in β' such that $(T',T) \in$ lock-completion(β'). But by the definition of lock-completion and the generic object well-formedness of β, $(T',T) \in$ lock-completion(β') if and only if $(T',T) \in$ lock-completion(β) = R. Thus the operations in ζ_1 are exactly those (T',v') that occur in β such that $(T',T) \in R$. Suppose (T_1,v_1) and (T_2v_2) are two operations in ζ_1 such that $(T_1,T_2) \in R$. By the definition of lock-completion, T_1 is lock-visible to T_2 in the longest prefix β_1 of β that does not include REQUEST_COMMIT(T_2,v_2). Applying Lemma 64 to β_1, (T_1,v_1) is in the intentions list of an ancestor of T_2 in the state s_1 reached by β_1, and by the effects of REQUEST_COMMIT(T_2,v_2), (T_1,v_1) precedes (T_2,v_2) in total(s_2,T_2), where s_2 is the state reached by β_1REQUEST_COMMIT(T_2,v_2). By Lemma 61, (T_1,v_1) precedes (T_2,v_2) in ζ_1. We conclude that the order of operations in ζ_1 is consistent with R.

If (T',v') and (T'',v'') are operations in ζ_1 and ζ_2 respectively, then we claim that $(T'',T') \notin R$. For if $(T'',T') \in R$, then since $(T',T) \in R$, by Lemma 62 we have also $(T'',T) \in R$. Then the characterization of ζ_1 above implies that (T'',v'') occurs in ζ_1, a contradiction. This claim implies that $\zeta_1\zeta_2$ is also a total ordering of Z' consistent with R. The induction hypothesis thus shows that perform(ξ') and perform($\zeta_1\zeta_2$) are equieffective serial object well-formed behaviors of S_X.

We show that (T,v) commutes with every operation (T'',v'') in ζ_2. There are two possibilities: either REQUEST_COMMIT(T'',v'') precedes REQUEST_COMMIT(T,v) in β, or vice versa. In the first case, let U denote the highest ancestor of T'' to which T'' is lock-visible in β'. By Lemma 64, $(T'',v'') \in$ s'.intentions(U), but by definition of ζ_2, U is not an ancestor of T. Therefore, by the preconditions for REQUEST_COMMIT(T,v), which is enabled in state s', we must have that (T,v) commutes with (T'',v''). In the second case, let β'' be the longest prefix of β not containing REQUEST_COMMIT(T'',v''), and let t be the state in which β'' leaves L_X. Also let U denote the highest ancestor of T to which T is lock-visible in β'', so that $(T,v) \in$ t.intentions(U). Now U is not an ancestor of T'', as otherwise $(T,T'') \in R$, contradicting the assumption that (T,v) is the last operation in ξ. Thus by the preconditions for REQUEST_COMMIT(T'',v''), (T'',v'') commutes with (T,v).

By the preconditions for REQUEST_COMMIT(T,v), which is enabled in state s', perform(total(s',T)(T,v)) = perform(ζ_1(T,v)) is a finite behavior of S_X, and it is clearly serial object well-formed, since β is generic object well-formed. We also showed above that perform($\zeta_1\zeta_2$) is a serial object well-formed behavior of S_X. Since (T,v) commutes with every operation in ζ_2, we have by Proposition 58 that perform($\zeta_1\zeta_2$(T,v)) is a serial object well-formed behavior of S_X. Since we saw that perform($\zeta_1\zeta_2$) is equieffective to perform(ξ'), and since perform(ξ) = perform(ξ'(T,v)) is clearly serial object well-formed, the definition of equieffectiveness implies that perform(ξ) is a behavior of S_X. This completes the proof that perform(ξ) is a serial object well-formed behavior of S_X.

Now let η be any other total ordering of Z that is consistent with R on the transaction component. Let η_1 and η_2 be the sequences of actions such that $\eta = \eta_1$(T,v)η_2. Then $\eta_1\eta_2$ is a total ordering of Z' consistent with R. The induction hypothesis shows that

perform($\eta_1\eta_2$) is a serial object well-formed behavior of S_X and that it is equieffective to perform(ξ'). Therefore, by Proposition 56, perform($\eta_1\eta_2$(T,v)) is equieffective to perform(ξ).

Part 1 applied to η implies that perform(η) is a serial object well-formed behavior of S_X; therefore, its prefix perform(η_1(T,v)) is also a serial object well-formed behavior of S_X.

By the characterization above for ζ_1, every operation in ζ_1 has transaction component that precedes T in R. Thus, since η is consistent with R, every operation in ζ_1 is contained in η_1, Thus, every operation in η_2 is contained in ζ_2, and so (T,v) commutes with every operation in η_2. Therefore perform(η) = perform(η_1(T,v)η_2) is equieffective to perform($\eta_1\eta_2$(T,v)), by Proposition 58.

Since perform(η) is equieffective to perform($\eta_1\eta_2$(T,v)) and perform($\eta_1\eta_2$(T,v)) is equieffective to perform(ξ), Lemma 57 implies that perform(η) is equieffective to perform(ξ), completing the proof.

Now we can prove that locking objects are locally dynamic atomic.

Proposition 66: L_X is locally dynamic atomic.

Proof: Let β be a finite generic object well-formed behavior of L_X and let T be a transaction name that is not a local orphan in β. We must show that local-pictures(β,T) $\subseteq$ **finbehs(S_X)**. So let Z be the set of operations occurring in β whose transactions are locally visible to T in β. Let ξ be a total ordering of Z consistent with local-completion(β) on the transaction components. We must prove that perform(ξ) is a behavior of S_X.

We claim that Z is allowable for β. To see this, suppose that (T',v') is an operation that occurs in Z. Then (T',v') occurs in β. Since T' is locally visible to T in β and T is not a local orphan in β, Lemma 50 implies that T' is not a local orphan in β. Now suppose that (T'',v'') is an operation that occurs in β and (T'',T') $\in$ lock-completion(β). Then T'' is lock-visible to T' in β, and hence, by Lemma 63, is locally visible to T' in β. Therefore, (T'',v'') is in Z.

We also claim that the ordering of ξ is consistent with lock-completion(β) on the transaction components. This is because the total ordering of ξ is consistent with local-completion(β), and Lemma 63 implies that locks-completion(β) is a subrelation of local-completion(β).

Lemma 65 then implies that perform(ξ) is a behavior of S_X, as needed.

Finally, we can show the main result of this section.

Theorem 67: L_X is dynamic atomic.

Proof: By Proposition 66 and Theorem 55.

An immediate consequence of Theorems 67 and the Dynamic Atomicity Theorem is that if $\mathcal{S}$ is a generic system in which each generic object is a locking object, then $\mathcal{S}$ is serially correct for all non-orphan transaction names.

10. Moss's Algorithm

In this section, we present Moss's algorithm for read-update locking [21] and its correctness proof. Once again, the algorithm is described as a generic system, and all that needs to be defined is the generic objects. We define the appropriate objects here, and show that they implement locking objects. It follows that they are dynamic atomic.

10.1. Moss Objects

For each object name X, we describe a generic object automaton M_X (a "Moss object"). The automaton M_X maintains a stack of "versions" of the corresponding serial object S_X, and manages "read locks" and "update locks".

The construction of M_X is based on a classification of all the accesses to X as either *read accesses* or *update accesses*. We assume that this classification satisfies the property that every operation (T,v) of a read access T is transparent. If ξ is a sequence of operations of X, we let *update*(ξ) denote the subsequence of ξ consisting of those operations whose first components are update accesses. Proposition 59 implies that if perform(ξ) is a serial object well-formed behavior of S_X, then perform(update(ξ)) is also a serial object well-formed behavior of S_X, and perform(update(ξ)) is equieffective to perform(ξ).

M_X has the usual action signature for a generic object automaton for X. A state s of M_X has components s.created, s.commit-requested, s.update-lockholders and s.read-lockholders, all sets of transactions, and s.map, which is a function from s.update-lockholders to states of the serial object automaton S_X. We say that a transaction in update-lockholders *holds an update-lock*, and similarly that a transaction in read-lockholders *holds a read-lock*. The start states of M_X are those in which update-lockholders = $\{T_0\}$ and map(T_0) is a start state of the serial object S_X, and the other components are empty.

If $\mathcal{U}$ is a finite set of transactions such that for all T and T' in $\mathcal{U}$, either T is an ancestor of T' or vice versa, then we define *least*($\mathcal{U}$) to be the unique transaction in $\mathcal{U}$ that is a descendant of all transactions in $\mathcal{U}$. Some of the following actions contain preconditions in which the function "least" is applied to the set s'.update-lockholders. In case least(s'.update-lockholders) is undefined, the precondition is assumed to be false.[19]

The transition relation of M_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'.update-lockholders
  then
    s.update-lockholders = (s'.update-lockholders - {T}) $\cup$ {parent(T)}

---

[19]In fact, in all states s' that arise in executions having generic object well-formed behaviors, least(s'.update-lockholders) is defined.

s.map(parent(T)) = s'.map(T)
s.map(U) = s'.map(U) for U ∈ s.update-lockholders - {parent(T)}
if T ∈ s'.read-lockholders
then s.read-lockholders = (s'.read-lockholders - {T}) ∪ {parent(T)}

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$
Effect:
s.update-lockholders = s'.update-lockholders - descendants(T)
s.read-lockholders = s'.read-lockholders - descendants(T)
s.map(U) = s'.map(U) for all U ∈ s.update-lockholders

REQUEST_COMMIT(T,v), T a read access to X
Precondition:
T ∈ s'.created - s'.commit-requested
s'.update-lockholders ⊆ ancestors(T)
there is a state t of S_X such that
(s'.map(least(s'.update-lockholders)),perform(T,v),t) is a move of S_X
Effect:
s.commit-requested = s'.commit-requested ∪ {T}
s.read-lockholders = s'.read-lockholders ∪ {T}

REQUEST_COMMIT(T,v), T an update access to X
Precondition:
T ∈ s'.created - s'.commit-requested
s'.update-lockholders ∪ s'.read-lockholders ⊆ ancestors(T)
there is a state t of S_X such that
(s'.map(least(s'.update-lockholders)),perform(T,v),t) is a move of S_X
Effect:
s.commit-requested = s'.commit-requested ∪ {T}
s.update-lockholders = s'.update-lockholders ∪ {T}
s.map(T) = t
s.map(U) = s'.map(U) for all U ∈ s.update-lockholders - {T}

When an access transaction is created, it is added to the set created. When M_X is informed of a commit, it passes any locks held by the transaction to the parent, and also passes any serial object state stored in map. When M_X is informed of an abort, it discards all locks held by descendants of the transaction. A response containing return value v to an access T can be returned only if the access has been created but not yet responded to, every holder of a conflicting lock is an ancestor of T, and perform(T,v) can occur in a move of S_X from the state that is the value of map at least(update-lockholders). When this response is given, T is added to commit-requested and granted the appropriate lock. Also, if T is a update access, the resulting state is stored as map(T), while if T is a read access, no change is made to map.

It is easy to see that M_X is a generic object, i.e, that it has the correct external signature and preserves generic object well-formedness. The following is also easy to prove, using induction of the length of a schedule.

Lemma 68: Let β be a finite schedule of M_X. Suppose that β can leave M_X in state s. Suppose T ∈ s.update-lockholders and T' ∈ s.read-lockholders ∪ s.update-

lockholders. Then either T is an ancestor of T' or else T' is an ancestor of T.

Note that it is permissible to classify all accesses as update accesses. The Moss object constructed from such a classification implements exclusive locking. Thus, the results we obtain about Moss objects also apply to exclusive locking as a special case.

10.2. Correctness Proof

In this subsection, we show that M_X is dynamic atomic. In order to show this, we produce a possibilities mapping from M_X to L_X as defined in Section , thereby showing that M_X implements L_X. Note that M_X is not describable as a simple special case of L_X: the two algorithms maintain significantly different data structures. Nevertheless, a possibilities mapping can be defined.

We begin by defining the mapping f. Let f map a state s of M_X to the set of states t of L_X that satisfy the following conditions.

1. s.created = t.created.
2. s.commit-requested = t.commit-requested.
3. s.read-lockholders is exactly the set of transaction names T such that t.intentions(T) contains a read operation.
4. s.update-lockholders is exactly the set of transaction names T such that t.intentions(T) contains an update operation, together with T_0.
5. For every transaction name T, perform(update(total(t,T))) is a finite behavior of S_X that can leave S_X in the state s.map(T'), where T' is the least ancestor of T such that T' $\in$ s.update-lockholders.

Lemma 69: f is a possibilities mapping from M_X to L_X.

Proof: The proof involves checking the conditions in the definition of a possibilities mapping. These checks are completely straightforward, but numerous and very tedious. For completeness, we include the details here, although the reader will probably not wish to read them.

It is easy to see that $t_0 \in f(s_0)$, where s_0 and t_0 are start states of M_X and L_X, respectively. Let s' and t' be reachable states of M_X and L_X, respectively, such that t' $\in$ f(s'). Suppose (s',π,s) is a step of M_X. We produce t such that (t',π,t) is a step of L_X and t $\in$ f(s). We proceed by cases.

1. π = CREATE(T), T an access to X.

   Since π is an input of L_X, π is enabled in state t'. Choose t so that (t',π,t) is a step of L_X. We show that t $\in$ f(s).

   The effects of π as an action of M_X and L_X imply that s.created = s'.created $\cup$ {T} and t.created = t'.created $\cup$ {T}. Moreover, all of the other components of s or t are identical to the corresponding components of s' or t', respectively. Since t' $\in$ f(s'), we have s'.created = t'.created, so that s.created = t.created, thus showing the first condition in the definition of f. The other conditions hold in s and t because they hold in s' and t' and none of the relevant

components are modified by π.

2. π = INFORM_COMMIT_AT(X)OF(U).

Since π is an input of L_X, π is enabled in state t'. Choose t so that (t',π,t) is a step of L_X. We show that t $\in$ f(s).

The first and second conditions hold in s and t because they hold in s' and t' and none of the relevant components are modified by π.

The effects of π as an action of L_X imply that t.intentions(W) = t'.intentions(W) unless W $\in$ {U,parent(U)}, t.intentions(parent(U)) = t'.intentions(parent(U))t'.intentions(U), and t.intentions(U) = λ. We consider two cases.

a. t'.intentions(U) contains a read operation.

Then the set of transaction names T such that t.intentions(T) contains a read operation is exactly the set of T such that t'.intentions(T) contains a read operation, with U removed and parent(U) added. Since t' $\in$ f(s'), s'.read-lockholders is exactly the set of transaction names T such that t'.intentions(T) contains a read operation; in particular, U $\in$ s'.read-lockholders. The effects of π as an action of M_X imply that s.read-lockholders = s'.read-lockholders - {U} $\cup$ {parent(U)}. Thus, s.read-lockholders is exactly the set of T such that t.intentions(T) contains a read operation.

b. t'.intentions(U) does not contain a read operation.

Then the set of transaction names T such that t.intentions(T) contains a read operation is exactly the set of T such that t'.intentions(T) contains a read operation. Since t' $\in$ f(s'), s'.read-lockholders is exactly the set of transaction names T such that t'.intentions(T) contains a read operation; in particular, U $\notin$ s'.read-lockholders. The effects of π as an action of M_X imply that s.read-lockholders = s'.read-lockholders. Thus, s.read-lockholders is exactly the set of T such that t.intentions(T) contains a read operation.

This shows the third condition. The proof of the fourth condition is analogous to that for the third condition.

Finally, fix some transaction T and let T' be the least ancestor of T such that T' $\in$ s.update-lockholders. The discussion is divided into subcases, depending on the relation between T and U in the transaction tree.

a. U is an ancestor of T.

Then total(t,T) = total(t',T). Let T'' be the least ancestor of T in s'.update-lockholders. Since t' $\in$ f(s'), perform(update(total(t',T))) is a finite behavior of S_X that can leave S_X in the state s'.map(T'').

If U = T'', then the effects of π as an action of M_X imply that s.update-lockholders = s'.update-lockholders - {T''} $\cup$ {parent(T'')}, so T' = parent(T''). Then s.map(T') = s.map(parent(T'')) = s'.map(T'').

If U $\neq$ T'' and U $\in$ s'.update-lockholders, then by definition of T'', U

is a strict ancestor of T''. Then s.map(T'') = s'.map(T'') and T'' = T', so again s.map(T') = s'.map(T'').

If U ≠ T'' and U is not in s'.update-lockholders, then s.update-lockholders = s'.update-lockholders and s.map = s'.map; thus, T'' = T' and so s.map(T') = s'.map(T'').

In each case, we have shown that s.map(T') = s'.map(T''); therefore, perform(update(total(t,T))) is a finite behavior of S_X that can leave S_X in the state s.map(T').

b. U is not an ancestor of T, but parent(U) is an ancestor of T.

If U ∈ s'.update-lockholders then Lemma 68 implies that no transaction in ancestors(T) - ancestors(parent(U)) can be in s'.update-lockholders ∪ s'.read-lockholders. The effects of π as an action of M_X therefore show that T' = parent(U). These effects also show that s.map(parent(U)) = s'.map(U). Since t' ∈ f(s'), t'.intentions(W) must be empty for all W ∈ ancestors(T) - ancestors(parent(U)). By the effects of π as an action of L_X, t.intentions(W) = t'.intentions(W) unless W equals U or parent(U), so t.intentions(W) is empty for all W ∈ ancestors(T) - ancestors(parent(U)). Thus, total(t,T) = total(t,parent(U)). The effects of π as an action of L_X also show that total(t,parent(U)) = total(t',U), so that total(t,T) = total(t',U). Since t' ∈ f(s') and U is the least ancestor of U in s'.update-lockholders, perform(update(total(t',U))) is a finite behavior of S_X that can leave S_X in state s'.map(U). The equalities we have proved show that perform(update(total(t,T))) is a finite behavior of S_X that can leave S_X in state s.map(T').

If U ∉ s'.update-lockholders then s.update-lockholders = s'.update-lockholders and s.map = s'.map. Thus, T' is the least ancestor of T in s'.update-lockholders, and s.map(T') = s'.map(T'). Since t' ∈ f(s'), there are no update operations in t'.intentions(U). Then the effects of π as an action of L_X imply that update(total(t,T)) = update(total(t',T)). Thus, perform(update(total(t,T))) = perform(update(total(t',T))), which is, by the fact that t' ∈ f(s'), a finite behavior of S_X that can leave S_X in state s'.map(T') = s.map(T').

c. parent(U) is not an ancestor of T.

The effects of π ensure that T' is the least ancestor of T in s'.update-lockholders, s.map(T') = s'.map(T') and total(t,T) = total(t',T). The result follows immediately from the fact that t' ∈ f(s').

This completes the demonstration of the fifth condition.

3. π = INFORM_ABORT_AT(X)OF(U).

Since π is an input of L_X, π is enabled in state t'. Choose t so that (t',π,t) is a step of L_X. We show that t ∈ f(s).

The first and second conditions hold in s and t because they hold in s' and t' and none of the relevant components are modified by π.

The effects of π as an action of L_X imply that t.intentions(W) = t'.intentions(W) unless W is a descendant of U, and t.intentions(W) = λ if W is a descendant of U. Thus, the set of transaction names T such that t.intentions(T) contains a read operation is exactly equal to the set of T such that t'.intentions(T) contains a read operation with the descendants of U removed. Similarly the effects of π as an action of M_X show that s.read-lockholders equals s'.read-lockholders with the descendants of U removed. Since t' $\in$ f(s'), the set of transaction names T such that t'.intentions(T) contains a read operation equals s'.update-lockholders. Thus, the set of T such that t.intentions(T) contains a read operation equals s.update-lockholders, as required. This shows the third condition. The proof of the fourth condition is analogous to that for the third condition.

Finally, fix some transaction T and let T' be the least ancestor of T such that T' $\in$ s.update-lockholders. The discussion is divided into subcases, depending on the relation between T and U.

a. U is an ancestor of T.

Then total(t,T) = total(t',parent(U)). The effects of π as an action of M_X imply that s.update-lockholders = s'.update-lockholders - descendants(U) and s.map(W) = s'.map(W) if W is not a descendant of U. Thus, T' is an ancestor of parent(U), and in fact must be the least ancestor of parent(U) in s'.update-lockholders. Since t' = f(s'), perform(update(total(t',parent(U)))) is a finite behavior of S_X that can leave S_X in state s'.map(T'). Thus, perform(update(total(t,T))) is a finite behavior of S_X that can leave S_X in state s.map(T').

b. U is not an ancestor of T.

The effects of π ensure that T' is the least ancestor of T in s'.update-lockholders, s.map(T') = s'.map(T') and total(t,T) = total(t',T). The result follows immediately from the fact that t' $\in$ f(s').

This completes the demonstration of the fifth condition.

4. π = REQUEST_COMMIT(U,u), U a read access to X.

We first show that π is enabled as an action of L_X in state t'. That is, we must show that U $\in$ t'.created - t'.commit-requested, that (U,u) commutes with every (V,v) in t'.intentions(U'), where U' $\notin$ ancestors(U), and that perform(total(t',U)(U,u)) is in finbehs(S_X).

Since t' $\in$ f(s'), t'.created = s'.created and t'.commit-requested = s'.commit-requested. Since π is enabled as an action of M_X in state s', we have that U $\in$ s'.created - s'.commit-requested. Therefore, U $\in$ t'.created - t'.commit-requested.

Suppose (in order to obtain a contradiction) that there exist U', V and v such that U' $\notin$ ancestors(U), (V,v) is in t'.intentions(U'), and (U,u) does not commute with (V,v). Since U is a read access and read accesses are transparent, Proposition 60 implies that either U = V or else V is a update access. Lemma 64 implies that U' is an ancestor of V, so that we cannot have V = U. Therefore, V is an update access. Since V is a update access and (V,v)

is in t'.intentions(U'), the fact that t' ∈ f(s') shows that U' ∈ s'.update-lockholders, Thus, since π is enabled in state s', U' is an ancestor of U. This is a contradiction; thus, we have shown that if U' is not an ancestor of U and (V,v) is in t'.intentions(U'), then (U,u) and (V,v) commute.

Finally, let U'= least(s'.update-lockholders). Since π is enabled in s', U' must be an ancestor of U and is thus the least ancestor of U in s'.update-lockholders. Therefore, the fact that t' ∈ f(s') implies that perform(update(total(t',U))) is a finite behavior of S_X that can leave S_X in state s'.map(U'). Since π is enabled in s', there is a move of S_X with behavior perform(U,u) starting from state s'.map(U'). Thus, perform(update(total(t',U)))perform(U,u) is a behavior of S_X. Since perform(update(total(t',U))) is equieffective to perform(total(t',U)), perform(total(t',U))perform(U,u) = perform(total(t',U)(U,u)) is in finbehs(S_X), since it is serial object well-formed.

Thus, π is enabled as an action of L_X in state t'. Choose t such that (t',π,t) is a step of L_X. We show that t ∈ f(s).

The effects of π imply that s.created = s'.created, t.created = t'.created, s.commit-requested = s'.commit-requested ∪ {U} and t.commit-requested = t'.commit-requested ∪ {U}. Since t' ∈ f(s'), we have t'.created = s'.created and t'.commit-requested = s'.commit-requested. Thus, s.created = t.created and s.commit-requested = t.commit-requested, so the first and second conditions hold.

The effects of π imply that s.read-lockholders = s'.read-lockholders ∪ {U}, t.intentions(U) = t'.intentions(U)(U,u), and t.intentions(W) = t'.intentions(W) for W ≠ U. Since t' ∈ f(s'), s'.read-lockholders is exactly the set of transaction names T such that t'.intentions(T) contains a read operation. Then s.read-lockholders = s'.read-lockholders ∪ {U}, which is exactly the set of transaction names T such that t.intentions(T) contains a read operation, so the third condition holds.

It is easy to see that the fourth condition holds in s and t, because it holds in s' and t' and the only relevant component that is modified is that t.intentions(U) = t'.intentions(U)(U,u), and (U,u) is a read operation.

For the final condition, consider any transaction T. Note that perform(update(total(t,T))) = perform(update(total(t',T))) and s.map = s'.map. Since the fifth condition holds in s' and t', it is easy to see that it holds in s and t.

5. π = REQUEST_COMMIT(U,u), U an update access to X.

We first show that π is enabled as an action of L_X in state t'. The proofs that U ∈ t'.created - t'.commit-requested and that perform(total(t',U)(U,u)) is in finbehs(S_X), are identical to the corresponding proofs for the read update case. We must show that (U,u) commutes with every (V,v) in t'.intentions(U'), where U' ∉ ancestors(U). We will show the stronger statement that if t'.intentions(U') is not the empty sequence, then U' ∈ ancestors(U). Since t' ∈ f(s'), if t'.intentions(U') is nonempty, then U' ∈ s'.read-lockholders ∪ s'.update-lockholders. Thus, since π is enabled as an action of M_X in state s',

U' ∈ ancestors(U).

Thus π is enabled as an action of L_X in state t'. Choose t such that (t',π,t) is a step of L_X. We show that t ∈ f(s). The first two conditions follow as for the read access case. The third condition holds in s and t because it holds in s' and t' and the only relevant component that is modified is that t.intentions(U) = t'.intentions(U)(U,u), and (U,u) is an update operation.

The effects of π imply that s.update-lockholders = s'.update-lockholders ∪ {U}, t.intentions(U) = t'.intentions(U)(U,u), and t.intentions(W) = t'.intentions(W) for W ≠ U. Since t' ∈ f(s'), s'.update-lockholders is exactly the set of transaction names T such that t'.intentions(T) contains an update operation, together with T_0. Thus, s.update-lockholders = s'.update-lockholders ∪ {U}, which is exactly the set of T such that t.intentions(T) contains an update operation, together with T_0. Thus, the fourth condition is satisfied.

Finally, we show the fifth condition. Fix any transaction name T. If T ≠ U, then since U is an access, T is not a descendant of U; then the fifth condition holds in s and t because if holds in s' and t' and none of the relevant components are modified. So suppose that T = U.

The effects of π as an action of M_X imply that s.map(U) is equal to some state r of S_X such that (s'.map(U'),perform(U,u),r) is a move of S_X, where U' = least(s'.update-lockholders); also, s.map(W) = s'.map(W) for all W ≠ U. Since all members of s'.update-lockholders must be ancestors of U by the preconditions of π in M_X, U' is the least ancestor of U in s'.update-lockholders, so the fact that t' ∈ f(s') implies that perform(update(total(t',U))) is a finite behavior of S_X that can leave S_X in state s'.map(U'). Thus, perform(update(total(t',U)))perform(U,u) is a finite behavior of S_X that can leave S_X in state s.map(U). But perform(update(total(t',U)))perform(U,u) = perform(update(total(t',U)(U,u))) = perform(update(total(t,U))). Thus, perform(update(total(t,U))) is a finite behavior of S_X that can leave S_X in state s.map(U), as required.

Proposition 70: M_X implements L_X.

Proof: By Lemma 69 and Theorem 3.

Theorem 71: M_X is dynamic atomic.

Proof: By Proposition 70 and Theorem 67.

An immediate consequence of Theorems 71, 67 and the Dynamic Atomicity Theorem is that if $\mathcal{S}$ is a generic system in which each generic object is either a Moss object or a locking object, then $\mathcal{S}$ is serially correct for all non-orphan transactions names.

11. Conclusions

We have presented a formal model for reasoning about atomic transactions that can include nested subtransactions, and have used it to carry out an extensive development of the important ideas about locking algorithms. First, we have stated the correctness conditions to be satisfied by

transaction-processing algorithms; we have stated these at the user interface to the transaction-processing system. Second, we have stated and proved a general Serializability Theorem that implies the correctness of transaction-processing algorithms. Third, we have defined the concept of "dynamic atomicity", a sufficient condition for satisfying the hypotheses of the Serializability Theorem. Fourth, we have presented two locking algorithms: a new general commutativity-based locking algorithm and a previously-known read-update locking algorithm. Fifth, we have provided complete correctness proofs for both algorithms. We have proved the general algorithm correct by showing that it satisfies the dynamic atomicity condition, and then we have proved the read-update algorithm correct by showing that it implements the general algorithm. All of these tasks have been quite manageable within the given framework.

The proofs we have constructed are very modular. Many interesting concepts are captured by formal definitions, and many facts about these concepts are captured by formally-stated lemmas. This modularity makes the development much easier to understand than it would be without it. Moreover, much of the machinery is reusable for presenting and verifying other algorithms.

We have already used our model to present and prove correctness of several other kinds of transaction-processing algorithms, including timestamp-based algorithms for concurrency control and recovery [1] and algorithms for management of replicated data [10] and of orphan transactions [14]. Our treatment of timestamp algorithms is especially noteworthy because it parallels the work in this paper very closely.

Briefly, the paper [1] contains descriptions of two timestamp algorithms: the timestamp algorithm of Reed [24], designed for data objects that are accessible only by read and write operations. and a new general algorithm that accommodates arbitrary data types. (This latter algorithm generalizes work of Herlihy [13] for single-level transactions.) These algorithms both involve assignment of ranges of timestamp values to transactions in such a way that the interval of a child transaction is included in the interval of its parent, and the intervals of siblings are disjoint. Responses to accesses are determined from previous accesses with earlier timestamps.

These algorithms are analyzed using the Serializability Theorem of this paper. This time, the sibling order used is the timestamp order. Now the view condition says that the processing of accesses to X is "consistent" with the timestamp order, in that reordering the processing in timestamp order yields a correct behavior for S_X. The Serializability Theorem implies that the timestamp algorithms are serially correct for all non-orphan transaction names. Again, each algorithm is described as the composition of object automata and a controller. Again, a local condition ("static atomicity") is defined, this time saying that an object satisfies the view condition using the timestamp order. As long as each object is static atomic, the whole system is serially correct for non-orphan transactions. Again, we have the flexibility to implement objects independently as long as static atomicity is guaranteed. We show that both algorithms ensure static atomicity.

There is much more that could be done using this model. For example, it would be interesting to model other kinds of locking algorithms, such as those using multigranularity locking [11], tree locking [2], and predicate locking [7]. Perhaps the dynamic atomicity and local dynamic atomicity conditions defined in this paper will prove useful for reasoning about these other algorithms as well.

It would be interesting to see if our Serializability Theorem can be used to prove correctness of other concurrency control algorithms besides those based on locking or timestamps.

It would also be interesting to integrate our approach more closely with the classical approach, to try to combine the advantages of both. Our framework is more general than the classical model, (because of its integrated treatment of concurrency control and recovery and because it allows transactions to nest). On the other hand, our model includes more detail than the classical model, and so it may seem more complicated. For example, the classical Serializability Theorem is stated in simple combinatorial terms, while our Serializability Theorem involves involves a fine-grained treatment of individual actions. We wonder if there is a simple combinatorial condition similar to the hypothesis of the classical theorem (but taking suitable account of nesting and failures), that implies the general correctness conditions described in this paper.

It would also be particularly interesting to use the framework to model some of the very complex transaction-processing algorithms that tolerate processor "crashes", i.e., failures that obliterate the contents of volatile memory [12].

References

[1] Aspnes, J. and Fekete, A. and Lynch, N. and Merritt, M. and Weihl, W.
A Theory of Timestamp-Based Concurrency Control for Nested Transactions.
In *Proceedings of 14th International Conference on Very Large Data Bases*. August 1988.
To appear.

[2] Bayer, R. and Schkolnick, M.
Concurrency of Operations on B-trees.
Acta Informatica 9:1-21, 1977.

[3] Beeri, C. and Bernstein, P.A. and Goodman, N.
A Model for Concurrency in Nested Transaction Systems.
Technical Report TR-86-03, Wang Institute, March, 1986.

[4] Beeri, C. and Bernstein, P. and Goodman, N. and Lai, M. and Shasha, D.
A Concurrency Control Theory for Nested Transactions.
In *Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pages 45-62. August, 1983.

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

[6] Davies, C.T.
Recovery Semantics for DB/DC System.
In *Proceedings of 28th ACM National Conference*, pages 136-141. 1973.

[7] Eswaran, K.P. and Gray, J.N. and Lorie, R.A. and Traiger, I.L.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.
Also published as IBM RJ1487, December 1974.

[8] Fekete, A. and Lynch, N. and Merritt, M. and Weihl, W.
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.

[9] Gawlick, D.
Processing Hot Spots in High Performance Systems.
In *Proceedings of 30th IEEE Computer Society International Conference*, pages 249-251. 1985.

[10] Goldman, K. and Lynch, N.
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.

[11] Gray, J. and Lorie, R. and Putzulo, A. and Traiger, J.
Granularity of Locks and Degrees of Consistency in a Shared Database.
Technical Report RJ1654, IBM, September, 1975.

[12] Gray, J. and Lorie, R. and Putzulo, A. and Traiger, J.
The Recovery manager of the System R Database Manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[13] Herlihy, M.
Extending Multiversion Time-Stamping Protocols to Exploit Type Information.
IEEE Transactions On Computers (C-36), April, 1987.

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

[15] Kung, H. and Robinson, J.
On Optimistic Methods for Concurrency Control.
ACM Transactions on Database Systems 6(2):213-226, June, 1981.

[16] Liskov, B.
Distributed Computing in Argus.
Communications of ACM 31(3):300-312, March, 1988.

[17] Lynch, N.
Concurrency Control for Resilient Nested Transactions.
Advances in Computing Research 3:335-373, 1986.

[18] Lynch, N.
I/O Automata: A Model for Discrete Event Systems.
Technical Memo MIT/LCS/TM-351, Massachusetts Institute Technology, Laboratory for Computer Science, March, 1988.
Also, in 22nd Annual Conference on Information Science and Systems, Princeton University, Princeton, N.J., March 1988.

[19] Lynch, N. and Merritt, M.
Introduction to the Theory of Nested Transactions.
In *International Conference on Database Theory*, pages 278-305. Rome, Italy, September, 1986.
Also, expanded version in MIT/LCS/TR-367 July 1986. Also, expanded version to appear in Theoretical Computer Science.

[20] Lynch, N. and Tuttle, M.
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.

[21] Moss, J.E.B.
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.

[22] Moss, J. and Griffeth, N. and Graham, M.
Abstraction in Concurrency Control and Recovery Management(revised).
Technical Report, University of Massachusetts at Amherst, May, 1986.
Technical Report COINS Technical Report 86-20.

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

[24] Reed, D.P.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, Massachusetts Institute Technology, 1978.
Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, Massachusetts Institute Technology, September 1978.

[25] Spector, A. and Swedlow, K.
Guide to the Camelot Distributed Transaction Facility: Release 1.
October, 1987
Available from Carnegie Mellon University, Pittsburgh, PA.

[26] Thomas, R.
A Majority Consenus Apprach to ConcurrencyControl for Multiple Copy Databases.
ACM Transactions on Database Systems 4(2):180-209, June, 1979.

[27] Weihl, W.E.
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.

[28] Weikum, G.
A Theoretical Foundation of Multi-Level Concurrency Control.
In *Proceedings of 5th ACM Symposium on Principles of Database Systems.* March, 1986.

[29] Yannakakis, M.
Serializability by Locking.
Journal of the ACM 31(2):227-244, 1984.

Table of Contents

OFFICIAL DISTRIBUTION LIST

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

2 copies

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

2 copies

Director, Code 2627
Naval Research Laboratory
Washington, DC 20375

6 copies

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

12 copies

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

2 copies

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

1 copy