Nested Transactions and Read/Write Locking

(Preliminary Report)

Alan Fekete[1]
Nancy Lynch[2]
Michael Merritt[3]
William Weihl[4]

1. Introduction

A major part of database research over several years has been the design and analysis of algorithms to maintain consistent data in the face of interleaved accesses, aborts of operations, replication of information and failures of system components. The most popular and simple protocol is two phase locking with separate read and write locks; other methods include arbitrary conflict-based locking, timestamp-based techniques, and locking that uses special structure of the data (e.g. a hierarchical arrangement) [Gr,T,KS,Ko,We]. A powerful theory has been developed to prove the correctness of these algorithms, based on the idea that a protocol is correct if it ensures that all executions are equivalent to serial executions [EGLT,P,BG]. This theory proves serializability by showing that a precedence graph contains no cycles.

Recently, some ideas in database system design and more general distributed system design have led several research groups to study the possibility of giving more structure to the transactions that are the basic unit of atomicity. When a transaction can contain concurrent operations that are to be performed atomically, or operations which can be aborted independently, we say that the operations form *subtransactions* of the original transaction. Thus we consider a system where transactions can be *nested*. This idea was first suggested by Davies under the name *spheres of control* [D]. A primitive example of this concept is implemented in System R, where a recovery block can be aborted and the transaction restarted at the last savepoint. In general distributed systems like Argus [LiS,LHJLSW] or Clouds [A], the basic services are often provided by Remote Procedure Calls which, at their best ("Only Once" semantics), are atomic. Since providing a service will often require using other services, the transactions that implement services ought to be nested.

The implementation of a nested transaction system requires extending the algorithms that have previously been considered for concurrency control, recovery and replication. The work of Reed [R] extended multi-version timestamp concurrency control to provide nested transaction data management. Moss [Mo] extended two phase locking with separate read and write locks to handle nesting, and this algorithm is the basis of data management in the Argus system implemented at MIT.

This paper is part of a major research effort to offer clean, readable descriptions of algorithms for managing data in a nested transaction system, together with rigorous proofs of the correctness of these algorithms. Other parts of the project include studying replicated data management algorithms, orphan elimination algorithms and general atomicity of abstract objects. All this work is based on a simple model of concurrent systems using *I/O automata* and an operational style of reasoning about their schedules. The first fruits of this program are detailed in [LM] which proves the correctness of exclusive locking, and which provides a basic framework for presenting the ideas of this paper.

This paper's contribution is threefold. First, it proves for the first time the correctness of Moss' algorithm, an algorithm which has been used in practice. Our discussion covers both concurrency control and recovery from aborts. However, we do not consider all the failure cases that the real system must deal with, as our model does not yet include crashes which compromise the system state. Second, we provide technical definitions (for *equieffectiveness* and *transparency*) which seem to capture exactly those properties of read operations that the algorithm depends on. Third, this paper provides another example of the power and value of the basic model of serial correctness first proposed in [LM], and of the operational style of reasoning with I/O automata.

In this paper we first review the I/O automaton model of computation. This is very similar to models like Communicating Sequential Processes [Ho], in that automata interact by synchronizing on shared operations. The main difference from other models is that we distinguish the input and output operations of each automaton. Any operation shared between components of a system can be an output of at most one component, and that component is in control of the operation, because no automaton is allowed to refuse to execute an input. Though automata have states as well as operations, we concentrate our analysis on the sequence of operations performed (the *schedule* of the system) - this operational mode of reasoning is quite different from assertional invariant methods used elsewhere in reasoning about distributed systems, but we find it very powerful and yet simple for the set of problems we consider.

Next, we show how to use I/O automata to model the parts of a nested transaction system. Each transaction is represented by an

[1] Department of Mathematics, Harvard University, Cambridge, Mass.

[2] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

[3] AT&T Bell Laboratories, Murray Hill, New Jersey

[4] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

The work of the second author was supported in part by the Office of Naval Research under Contract N00014 85 K 0168, by the Office of Army Research under contract DAAG29 84 K 0058, by the National Science Foundation under Grants MCS 8306854, DCR 83-02391, and 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 DCR-8510014, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014 83-K 0125.

automaton, as is each data object The actions of calling a subtransaction, invoking an access to an object, and returning a result are each split into two operations, one requesting the action and one delivering the request to the recipient The request operation is an output of the caller and an input to the scheduler (which acts as a communication system) while the delivery operation is an output of the scheduler and an input of the recipient Thus, each transaction (and each object) shares operations only with the scheduler A *serial system* is the result of composing transaction and object automata with a *serial scheduler*, which runs the subtransactions of any transaction sequentially (with no concurrency between siblings) and only aborts transactions before they start running The serial scheduler is very simple to understand and is used as the basis of our correctness condition

We then introduce a *R/W Locking system* to model a system using Moss' locking algorithm to manage data We use a new sort of I/O automaton called a *R/W Locking object* which is like the object automaton of the serial system, but which maintains lock tables and versions of the object so that it can respond correctly when aborts occur It also delays operations until it is permitted to respond by the locking rules We also use a new sort of scheduler called a *generic scheduler*, which transmits requests to the appropriate recipient with arbitrary delay, allowing siblings to run concurrently or to abort after performing some work A R/W Locking system is the result of composing the transaction automata, R/W Locking objects and generic scheduler

A R/W Locking system allows more concurrency than a serial system, but it is correct in the sense (first suggested in [LM]) that each transaction which does not have an aborted ancestor is unable to tell whether it is running in a R/W Locking system or in a serial system The proof of this correctness condition is the main result of this paper

The proof proceeds by taking an arbitrary schedule of a R/W Locking system (a *concurrent* schedule) and explicitly showing how to rearrange the operations to get a schedule of the serial system The permitted rearrangements (which do not alter the sequence of events at any transaction) are those that are *write-equivalent* to the original sequence

A key contribution of this paper is in identifying exactly the properties of read and write accesses which are required to guarantee correctness of Moss' algorithm Write accesses require no special properties However, it is necessary that read accesses leave the object in "essentially" the same state as they found it We define *equieffective* schedules to be those that leave the object in "essentially" the same state, where "essentially" means "as far as later operations can detect" Then an object schedule with a read access appended is required to be equieffective to the same schedule without the read access

There have been several other attempts to provide rigorous proofs of the correctness of algorithms for data management in nested transaction systems The first was [Ly], which presented a model that successfully handled exclusive locking, but which proved difficult to extend to more complicated problems such as orphan elimination [Go] The main deficiencies of this earlier model seem to be the lack of distinction between inputs and outputs, and the lack of explicit representations for transactions and their interfaces These deficiencies were remedied in [LM], where the operational model discussed above was defined, this paper again proved correctness of exclusive locking This paper continues the work of [LM] by dealing with an algorithm with separate read and write locks (The result of this paper implies a main result of [LM], since when no accesses are distinguished as read accesses, Moss' algorithm degenerates into exclusive locking) A different program to study concurrency control in nested transaction systems has been offered in [BBGLS,BBG], where a major motivation is to analyze protocols that operate on data at different levels of abstraction, but where recovery is not considered The argument for the correctness of Moss' algorithm in [BBG] considers only the locking rules and not the state maintenance methods, so correctness is proved only in the absence of aborts Concurrency control and recovery algorithms are also analyzed in [MGG], but [MGG] is also concerned mainly with levels of abstraction

This paper uses many concepts from [LM], but we have repeated everything needed to make it self-contained, and indicated where definitions or details differ In Section 2, we review the model of I/O automata of [LT,LM] In Section 3, we define the automata that make up the serial system, namely the transaction automata, the basic object automata and the serial scheduler In Section 4, we specify the semantic conditions that read accesses must satisfy, using the technical notion of equieffective schedules In Section 5 we define the automata of the R/W Locking system, namely the R/W Locking objects (which have code based immediately on the algorithm of [Mo]) and the generic scheduler, and prove the main lemmas that relate the schedules of R/W Locking objects to the schedules of the basic objects Finally in Section 6 we prove that R/W Locking systems are serially correct at transactions no ancestor of which has aborted, and in particular at the root transaction which represents the external environment

2. I/O Automata

The following is a brief introduction to a model which is described in [LM] and developed at length, with extensions to express infinite behavior, in [LT]

All components in our systems, transactions, objects and schedulers, will be modelled by *I/O automata* An I/O automaton $\mathcal{A}$ has a set of *states*, some of which are designated as *initial states* It has *operations*, each classified as either an *input operation* or an *output operation* Finally, it has a transition relation, which is a set of triples of the form (s',π,s), where s' and s are states, and π is an operation This triple means that in state s', the automaton can atomically do operation π and change to state s An element of the transition relation is called a *step* of the automaton The output operations are intended to model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton

Given a state s' and an operation π, we say that π is *enabled* in s' if there is a state s for which (s',π,s) is a step We require the following condition

Input Condition Each input operation π is enabled in each state s'

This condition says that an I/O automaton must be prepared to receive any input operation at any time

An *execution* of $\mathcal{A}$ is an finite alternating sequence s_0,π_1, s_1,π_2, $,\pi_n,s_n$ of states and operations of $\mathcal{A}$, beginning and ending with a state Furthermore, s_0 is a start state of $\mathcal{A}$, and each triple (s',π,s) which occurs as a consecutive subsequence is a step of $\mathcal{A}$ From any execution, we can extract the *schedule*, which is the subsequence of the execution consisting of operations only Because transitions to different states may have the same operation, different executions may have the same schedule We say that a schedule α of $\mathcal{A}$ *can leave* $\mathcal{A}$ *in* state s if there is some execution of $\mathcal{A}$ with schedule α and final state s We say that an operation π is *enabled after* a schedule α of $\mathcal{A}$ if there exists a state s such that α can leave $\mathcal{A}$ in state s and π is enabled in s Since the same operation may occur several times in an execution or schedule, we refer to a single occurrence of an operation as an *event*.

We describe systems as consisting of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata, also. Thus, we define a composition operation for I/O automata, to yield a new I/O automaton. A set of I/O automata may be composed to create a *system* S, if the sets of output operations of the various automata are pairwise disjoint. (Thus, every output operation in S will be triggered by exactly one component.) A state of the composed automaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. The operations of the composed automaton are those of the component automata. Thus, each operation of the composed automaton is an operation of a subset of the component automata. An operation is an output of the composed automaton exactly if it is an output of some component. (The output operations of a system are intended to be exactly those that are triggered by components of the system, while the input operations of a system are those that are triggered by the system's environment.) During an operation π of a composed automaton, each of the components which has operation π carries out the operation, while the remainder stay in the same state.

An *execution* or *schedule* of a system is defined to be an execution or schedule of the automaton composed of the individual automata of the system. If α is a schedule of a system with component $\mathcal{A}$, then we denote by $\alpha|\mathcal{A}$ the subsequence of α containing all the operations of $\mathcal{A}$. Clearly, $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$.

The following lemma from [LM] expresses formally the idea that an operation is under the control of the component of which it is an output.

> **Lemma 1** Let α' be a schedule of a system S, and let $\alpha = \alpha'\pi$, where π is an output operation of component $\mathcal{A}$. If $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$, then α is a schedule of S.

We say that automaton $\mathcal{A}$ *preserves* a property P of schedules of $\mathcal{A}$ if $\alpha = \alpha'\pi$ satisfies P whenever α is a schedule $\mathcal{A}$, α' satisfies P and π is an output of $\mathcal{A}$.

3. Serial Systems

In this paper we define two kinds of systems: "serial systems" and "R/W Locking systems". Serial systems describe serial execution of transactions. They are defined for the purpose of giving a correctness condition for other systems, namely that the schedules of another svstem should look like schedules of the serial system to the transactions. As with serial executions of single-level transaction systems, serial systems are too inefficient to use in practice. Thus, we define R/W Locking systems, which allow transactions to run concurrently or abort after performing some work; these systems use Moss' algorithm to maintain locks and enough information to restore the states of objects after aborts occur.

In this section of the paper we define serial systems, which consist of transactions and basic objects communicating with a *serial scheduler*. Transactions and basic objects describe user programs and data, respectively. The serial scheduler controls communication between the other components, and thereby controls the orders in which the transactions create children or access data. All the system components are modelled as I/O automata. Most of this section is taken from [LM], with slight modifications to accomodate slight changes in definitions.

We represent the pattern of transaction nesting, which we call a *system type*, by a set of transaction names, organized into a tree by the mapping "parent()", with T_0 as the root. In referring to this tree, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendant. (A transaction is its own ancestor and descendant.) The leaves of this tree are called *accesses*. The accesses are partitioned, where each element of the partition contains the accesses to a particular object. The tree structure can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In any particular execution, however, only some of these transactions will actually take steps. We imagine that the tree structure is known in advance by all components of a system. The tree will, in general, be an infinite structure with infinite branching.

The classical transactions of concurrency control theory (without nesting) appear in our model as the children of a "mythical" transaction, T_0, the root of the transaction tree. (In work on nested transactions, such as Argus, the children of T_0 are often called "top-level" transactions.) It is very convenient to introduce the new root transaction to model the environment in which the rest of the transaction system runs. Transaction T_0 has operations that describe the invocation and return of the classical transactions. It is natural to reason about T_0 in the same way as about all of the other transactions. The only transactions which actually access data are the leaves of the transaction tree, and thus they are distinguished as "accesses". The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly.

We also assume that a system type includes a designated set V of *values*, to be used as return values of transactions.

A serial system of a given system type is the composition of a set of I/O automata. This set contains a transaction automaton for each *internal* (i.e. non-leaf, non-access) node of the transaction tree, a basic object automaton for each object, and a serial scheduler. These automata are described below.

3.1 Transactions

This paper differs from other work such as [BBG] in that we model the transactions explicitly. A non-access *transaction* T is modelled as an I/O automaton, with the following operations:

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

The CREATE input operation "wakes up" the transaction. The REQUEST_CREATE output operation is a request by T to create a particular child transaction.[6] The REPORT_COMMIT input operation reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The REPORT_ABORT input operation reports to T the unsuccessful completion of one of its children, without returning any other information. We call REPORT_COMMIT(T',v), for any v, and REPORT_ABORT(T') *report* operations for transaction T'. The REQUEST_COMMIT operation is an announcement by T that it has finished its work, and includes a value recording the results of that work.

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

It is convenient to use two separate operations, REQUEST_CREATE and CREATE, to describe what takes place when a subtransaction is activated The REQUEST_CREATE is an operation of the transaction's parent, while the actual CREATE takes place at the subtransaction itself In actual systems such as Argus, this separation does occur, and the distinction will be important in our results and proofs Similar remarks hold for the REQUEST_COMMIT and COMMIT operations[7] We leave the executions of particular transaction automata largely unspecified, the choice of which children to create, and what value to return, will depend on the particular implementation For the purposes of the schedulers studied here, the transactions (and in large part, the objects) are "black boxes" Nevertheless, it is convenient to assume that schedules of transaction automata obey certain syntactic constraints We therefore require that all transaction automata preserve well-formedness, as defined in the next paragraph We do not constrain the operation of a transaction automaton after schedules that violate well-formedness, but we will prove later that, when placed in any of the systems we consider, a transaction generates only well-formed schedules

We recursively define *well-formedness* for sequences of operations of transaction T Namely, the empty schedule is well-formed Also, if $\alpha = \alpha'\pi$ is a sequence of operations of T, where π is a single event, then α is well-formed provided that α' is well-formed, and the following hold

- If π is CREATE(T), then
  (i) there is no CREATE(T) event in α'

- If π is REPORT_COMMIT(T',v) for a child T' of T, then
  (i) REQUEST_CREATE(T') appears in α' and
  (ii) there is no REPORT_ABORT(T') event in α' and
  (iii) there is no REPORT_COMMIT(T',v') event with v'≠v in α'

- If π is REPORT_ABORT(T') for a child T' of T, then
  (i) REQUEST_CREATE(T') appears in α' and
  (ii) there is no REPORT_COMMIT event for T' in α'

- If π is REQUEST_CREATE(T') for a child T' of T, then
  (i) there is no REQUEST_CREATE(T') in α' and
  (ii) there is no REQUEST_COMMIT event for T in α' and
  (iii) CREATE(T) appears in α'

- If π is a REQUEST_COMMIT for T, then
  (i) there is no REQUEST_COMMIT event for T in α' and
  (ii) CREATE(T) appears in α'

These restrictions are very basic, they simply say that a transaction does not get created more than once, does not receive conflicting information about the fates of its children, and does not receive information about the fate of any child whose creation it has not requested, also, a transaction does not perform any output operations before it has been created or after it has requested to commit, and does not request the creation of the same child more than once Except for these minimal conditions, there are no a priori restrictions on allowable transaction behavior

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

Lemma 2 Let α be a well-formed sequence of operations of transaction T Then the following conditions hold

1 The first event in α is a CREATE(T) event, and there are no other CREATE events

2 If a REQUEST_COMMIT event for T occurs in α, then there are no later output events of T in α

3 There is at most one REQUEST_CREATE(T') event for each child T' of T, in α

4 There are not two different report operations in α for any child T' of T (However, there may be several events which are repeated instances of a single report operation)

5 Any report event for a child T' of T is preceded by REQUEST_CREATE(T') in α

Conversely, any sequence of operations of T satisfying these conditions is well-formed

3 2 Basic Objects

Recall that I/O automata are associated with non-access transactions only Since access transactions model abstract operations on shared data objects, we associate a single I/O automaton with each object, rather than one for each access The operations for each object are just the CREATE and REQUEST_COMMIT operations for all the corresponding access transactions Although we give these operations the same names as the operations of non-access transactions, it is helpful to think of the operations of access transactions in other terms also a CREATE corresponds to an invocation of an operation on the object, while a REQUEST_COMMIT corresponds to a response by the object to an invocation Actually, these CREATE and REQUEST_COMMIT operations generalize the usual invocations and responses in that our operations carry with them a designation of the position of the access in the transaction tree Thus, a *basic object* X is modelled as an automaton, with the following operations

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

As with transactions, while specific objects are left largely unspecified, it is convenient to require that schedules of basic objects satisfy certain syntactic conditions We recursively define ***well-formedness*** for sequences of operations of basic objects Namely, the empty schedule is well-formed Also, if $\alpha = \alpha'\pi$ is a sequence of operations of basic object X, where π is a single event, then α is well-formed provided that α' is well-formed, and the following hold

- If π is CREATE(T), then
  (i) there is no CREATE(T) event in α'

- If π is REQUEST_COMMIT for T, then
  (i) there is no REQUEST_COMMIT event for T in α', and
  (ii) CREATE(T) appears in α'

These restrictions simply say that the same access does not get created more than once, and that a basic object does not respond more than once to any access, and only responds to accesses that

[7] Note that we do not include a REQUEST_ABORT operation for a transaction we do not model the situation in which a transaction decides that its own existence is a mistake Rather we assign decisions to abort transactions to another component of the system the scheduler In practice, the scheduler must have some power to decide to abort transactions, as when it detects deadlocks or failures In Argus, transactions are permitted to request to abort, we regard this request simply as a "hint" to the scheduler to restrict its allowable executions in a particular way

have previously been created These requirements constrain the environment of the object slightly less than those in [LM], the added freedom makes some of the arguments slightly simpler We require that every basic object preserve well-formedness (this is a simple syntactic condition) The following easy lemma summarizes the properties of well-formed sequences of basic object operations

Lemma 3 Let α be a well-formed sequence of operations of basic object X Then for any access T to X, α contains one of the following

1 no CREATE(T) and no REQUEST_COMMIT(T,v) events, or

2 one CREATE(T) and no REQUEST_COMMIT(T,v) events, or

3 one CREATE(T) event and following that one REQUEST_COMMIT(T,v) event for some v

Conversely, any α satisfying this condition is well-formed

If α is a well-formed sequence of operations of X and T is an access to X such that α contains CREATE(T) but no REQUEST_COMMIT(T,v), we say that T is *pending* in α

3 3 Serial Scheduler

The third kind of component in a serial system is the serial scheduler The serial scheduler is also modelled as an automaton Whereas the transactions and basic objects have been specified to be any I/O automata whose operations and behavior satisfy simple syntactic restrictions, the serial scheduler is a fully specified automaton, particular to each system type It runs transactions according to a depth-first traversal of the transaction tree The serial scheduler can choose nondeterministically to abort any transaction 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) operation are that transaction T was never created Each child of T whose creation was requested must be either aborted or run to commitment with no siblings overlapping its execution, before T can commit The operations of the serial scheduler are as follows

Input Operations
 REQUEST_CREATE(T)
 REQUEST_COMMIT(T,v)
Output Operations
 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), $T \neq T_0$

The REQUEST_CREATE and REQUEST_COMMIT inputs are intended to be identified with the corresponding outputs of transaction and object automata, and correspondingly for the CREATE, REPORT_COMMIT and REPORT_ABORT output operations The COMMIT and ABORT operations are internal, marking the point in time where the decision on the fate of the transaction is irrevocable We call COMMIT(T) and ABORT(T) *return* operations for T

Each state s of the serial scheduler consists of six sets, named with record notation s create_requested, s created, s commit_requested, s committed, s aborted and s returned The set s commit_requested is a set of (transaction,value) pairs The others are sets of transactions There is exactly one initial state, in which the set create_requested is $\{T_0\}$, and the other sets are empty

The transition relation consists of exactly those triples (s',π,s) satisfying the pre- and postconditions below, where π is the indicated operation For brevity, we include in the postconditions only those conditions on the state s which may change with the operation If a component of s is not mentioned in the postcondition, it is implicit that the set is the same in s' and s

REQUEST_CREATE(T)
Postcondition
 s create_requested $=$ s' create_requested $\cup \{T\}$

REQUEST_COMMIT(T,v)
Postcondition
 s commit_requested $=$ s' commit_requested $\cup \{(T,v)\}$

CREATE(T)
Precondition
 $T \in$ s' create_requested - (s' created $\cup$ s' aborted)
 siblings(T) $\cap$ s' created $\subseteq$ s' returned
Postcondition
 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' returned
 children(T) $\cap$ s' create_requested $\subseteq$ s' returned
Postcondition
 s committed $=$ s' committed $\cup \{T\}$
 s returned $=$ s' returned $\cup \{T\}$

ABORT(T), $T \neq T_0$
Precondition
 $T \in$ s' create_requested - (s' created $\cup$ s' aborted)
 siblings(T) $\cap$ s' created $\subseteq$ s' returned
Postcondition
 s aborted $=$ s' aborted $\cup \{T\}$
 s returned $=$ s' returned $\cup \{T\}$

REPORT_ABORT(T), $T \neq T_0$
Precondition
 $T \in$ s' aborted

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition
 $T \in$ s' committed
 $(T,v) \in$ s' commit_requested

The input operations, REQUEST_CREATE and REQUEST_COMMIT, simply result in the request being recorded A CREATE operation can only occur if a corresponding REQUEST_CREATE has occurred and the CREATE has not already occurred The second precondition on the CREATE operation says that the serial scheduler does not create a transaction until all its previously created sibling transactions have returned That is, siblings are run sequentially The precondition on the COMMIT operation says that the scheduler does not allow a transaction to commit until its children have returned The precondition on the ABORT operation 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 (purely internal) commit or abort has occurred In particular, siblings might run in one order and be reported to their parent in the opposite order

One significant difference between our serial scheduler and the one in [LM] is that there the return operation and the report to the parent of the return are combined as a single operation, giving the parent the extra information of the order in which its children are run

The next lemma relates a schedule of the serial scheduler to the state which results from applying that schedule

Lemma 4 Let α be a schedule of the serial scheduler, and let s be a state which can result from applying α to the initial state Then the following conditions are true

1 T is in s create_requested exactly if T $= T_0$ or α contains a REQUEST_CREATE(T) event

2 T is in s created exactly if α contains a CREATE(T) event

3 (T,v) is in s commit_requested exactly if α contains a REQUEST_COMMIT(T,v) event

4 T is in s committed exactly if α contains a COMMIT(T) event

5 T is in s aborted exactly if α contains an ABORT(T) event

6 s returned = s committed $\cup$ s aborted

7 s committed $\cap$ s aborted = $\emptyset$

3 4 Serial Systems and Serial Schedules

The composition of transactions with basic objects and the serial scheduler for a given system type is called a ***serial system,*** and its operations and schedules are called ***serial operations*** and ***serial schedules*** respectively A sequence α of serial operations is said to be *well formed* provided that its projection at every transaction and basic object is well-formed

Lemma 5 Let α be a serial schedule Then α is well-formed

Proof A straightforward induction on the length of schedules The inductive step involves a case analysis based on the possible operations □

If α is a sequence of serial operations and T is a transaction such that α contains CREATE(T) but no return event for T, we say that T is *live* in α The following lemma states that only related transactions can be live concurrently, in a serial schedule

Lemma 6 Let α be a serial schedule, and T and T' transactions each of which is live in α Then either T is an ancestor of T' or T' is an ancestor of T

In order to talk about schedules, we introduce some terms to describe the fate of transactions Let α be any sequence of operations (We will use these same terms later for schedules of R/W Locking systems, so we make the definitions for general sequences) If T is a transaction and T' an ancestor of T, we say that T is *committed* to T' in α if COMMIT(U) occurs in α for every U which is an ancestor of T and a proper descendant of T' If T and T' are transactions we say that T is ***visible*** to T' in α if T is committed to lca(T,T') If π is one of the operations CREATE(T), REQUEST_CREATE(T'), COMMIT(T'), ABORT(T'), REPORT_COMMIT(T',v'), REPORT_ABORT(T',v') or REQUEST_COMMIT(T,v) where T' is a child of T, then we define *transaction(π)* to be T If T is a non-access transaction then the operations π with transaction(π) = T are the operations of the *automaton T together with the return operations for children of* T We denote by visible(α,T) the subsequence of α consisting of events π with transaction(π) to T in α Notice that every operation occurring in visible(α,T) is a serial operation

We collect here some straightforward consequences of these definitions

Lemma 7 Let α be a sequence of operations, and T, T' and T" transactions

1 If T is an ancestor of T', then T is visible to T' in α

2 T' is visible to T in α if and only if T' is visible to lca(T,T') in α

3 If T" is visible to T' in α and T' is visible to T in α, then T" is visible to T in α

4 If T' is a proper descendant of T, T" is visible to T' in α, but T" is not visible to T in α, then T" is a descendant of the child of T which is an ancestor of T'

Lemma 8 Let α and β be sequences of operations, such that β consists of a subset of the events of α

1 If transaction T is visible to transaction T' in β, then T is visible to T' in α

2 If event π is in visible(β,T), then π is in visible(α,T)

Lemma 9 Let α be a sequence of operations, and let T and T' be transactions Then visible(α,T)|T' is equal to α|T if T' is visible to T in α, and is equal to the empty sequence otherwise

Lemma 10 Let α be a sequence of operations Let T, T' and T" be transactions such that T" is visible to T' and to T in α Then T" is visible to T' in visible(α,T)

Lemma 11 Let T be a transaction, and let $\alpha\pi$ be a sequence of operations, where π is a single event

1 If transaction(π) is not visible to T in $\alpha\pi$, then visible($\alpha\pi$,T) = visible(α,T)

2 If transaction(π) is visible to T in $\alpha\pi$ and if π is not a COMMIT event, then visible($\alpha\pi$,T) = visible(α,T)π

3 If transaction(π) is visible to T in $\alpha\pi$, and π is COMMIT(U), then the events in visible($\alpha\pi$,T) are those visible in α to either T or U, together with π itself

Lemma 12 Let α be a well-formed sequence, and T any transaction Then visible(α,T) is well-formed

The next two lemmas are taken from [LM] (There, they are proved with slightly different definitions, but the essentially the same proofs work here)

Lemma 13 Let α be a serial schedule and T a transaction Then visible(α,T) is a serial schedule

Lemma 14 Let α be a serial schedule and T a transaction Let β = visible(α,T) Then $\gamma = \beta(\alpha - \beta)$ is a serial schedule

Let α be any sequence of operations If T is a transaction we say T is an *orphan* in α if ABORT(U) occurs in α for some ancestor U of T

3 5 Serial Correctness

We use serial schedules as the basis of our correctness definition, which was first given in [LM] Namely, we say that a sequence of operations is ***serially correct for a transaction*** *T* provided that its projection on T is identical to the projection on T of some serial schedule That is, the sequence "looks like" a serial schedule to

T Later in this paper we will define "R/W Locking systems" and show that their schedules are serially correct for every non-orphan transaction, and in particular that these schedules are serially correct for the root transaction T_0

Motivation for our use of serial schedules to define correctness derives from the simple behavior of the serial scheduler, which determines the sequence of interactions between the transactions and objects We believe the depth-first traversal of the transaction tree to be a natural notion of correctness which corresponds precisely to the intuition of how nested transaction systems ought to behave Furthermore, it is a natural generalization of serializability, the correctness condition generally chosen for classical transaction systems Serial correctness for T is a condition which guarantees to implementors of T that their code will encounter only situations which can arise in serial executions Correctness for T_0 is a special case which guarantees that the external world will encounter only situations which can arise in serial executions

It would be best if every transaction (whether an orphan or not) saw consistent data Ensuring this requires a much more intricate scheduler than the simple R/W Locking systems we describe In [HLMW], we describe and prove correctness of several algorithms for maintaining correctness for orphan transactions

Our approach is an example of a general technique for studying system algorithms A simple, intuitive and inefficient algorithm (automaton) is used to specify an acceptable collection of schedules for the system component The actual system component is more efficient or robust, but provides the same user interface The user is guaranteed that applications (transactions, in our work) which work well when run with the simple algorithm will work the same way when run with the actual system

4. Semantic Conditions

In the serial systems to be considered in this paper, accesses are classified as either *read* or *write* accesses In this section, we state the properties which these accesses are required to satisfy First, we define the fundamental concept of "equieffectiveness" of schedules, which is in turn used to define "transparency" of operations, an operation is said to be transparent if later accesses to the same object return values which are the same as in the situation where the operation did not occur We then prove certain consequences of these definitions, which will be used in the ensuing proofs Finally, we use the notion of transparency to specify the precise semantic conditions which read and write accesses must satisfy

4 1 Equieffective Schedules

We introduce the concept of equieffective schedules of a basic object X, in order to define precisely what schedules we will regard as "essentially" the same Intuitively, these are schedules which leave the automaton in states which are the same However, we are really interested in schedules, not states, so it is enough that they be indistinguishable by later operations

Formally, given two well-formed sequences α and β of operations of X, we say that α is *equieffective* to β if for every sequence ϕ of operations of X such that both $\alpha\phi$ and $\beta\phi$ are well-formed, $\alpha\phi$ is a schedule of X if and only if $\beta\phi$ is a schedule of X

Notice that if neither α nor β is a schedule of X, then α is trivially equieffective to β Also, notice that if α is equieffective to β and β is a schedule of X, then α is a schedule of X In the sense of semantic theory, equieffective schedules pass the same tests, where a test involves determining if a given sequence of operations can occur after the sequence being tested We limit the tests to sequences which do not violate well-formedness, for technical reasons, because we have not required the objects to behave sensibly if the inputs violate well-formedness Clearly, α is equieffective to β if and only if β is equieffective to α and in this case we say that α and β are equieffective sequences We have a restricted form of transitivity

Lemma 15 Let α, β and γ be sequences of operations of X such that the events in β are a subset of the events in α and the events in γ are a subset of the events in β (perhaps in different orders) If α and β are equieffective and also β and γ are equieffective, then α and γ are equieffective

We also have an extension result

Lemma 16 If α and β are equieffective sequences of operations of X which contain the same events, and ϕ is a sequence of operations of X such that $\alpha\phi$ is a well-formed schedule of X, then $\beta\phi$ is a well-formed schedule of X which is equieffective to $\alpha\phi$

We say that an operation π of basic object X is *transparent* if for any well-formed schedule $\alpha\pi$ of X, $\alpha\pi$ is equieffective to α Thus, later operations which do not violate well-formedness cannot detect whether π happened (Notice that we only require π to be undetectable in situations where it can occur, i e when $\alpha\pi$ is a schedule)

Lemma 17: Let α be a well-formed schedule of basic object X, and S a set of accesses to X such that any operation of a transaction in S that occurs in α is transparent Let β be the subsequence of α obtained by removing all the operations of accesses in S Then β is a well-formed schedule of X which is equieffective to α

4 2 Reordering and Combining Serial Schedules

In this subsection, we describe ways in which serial schedules can be modified and combined to yield other serial schedules These lemmas are used in the proof of Lemma 33, in Section 6 3 The first generalizes a lemma in [LM], taking into account the special properties of transparent operations The second is essentially the same as a lemma of [LM]

Lemma 18 Let $\alpha\beta_1$COMMIT(T') and $\alpha\beta_2$ be two serial schedules and T, T' and T" three transactions such that the following conditions hold

1 T' is a child of T" and T is a descendant of T" but not of T',

2 $\alpha\beta_1$ = visible($\alpha\beta_1$,T'),

3 $\alpha\beta_2$ = visible($\alpha\beta_2$,T),

4 α = visible($\alpha\beta_1$,T") = visible($\alpha\beta_2$,T") and

5 if any basic object has an output operation in β_2 then all its operations in β_1 are transparent

Then $\alpha\beta_1$COMMIT(T')β_2 is a serial schedule

Proof Straightforward by induction on the length of prefixes of $\alpha\beta_1$COMMIT(T')β_2 □

Lemma 19. Let αABORT(T') and $\alpha\beta$ be two serial schedules, and let T, T' and T" be transactions, such that the following conditions hold

1 T' is a child of T" and T is a descendant of T" but not of T',

2 $\alpha\beta$ = visible($\alpha\beta$,T), and

3 $\alpha = \text{visible}(\alpha,T'') = \text{visible}(\alpha\beta,T'')$

Then αABORT(T')β is a serial schedule

4 3 Semantics of Read Accesses

Finallv, we are ready to state the conditions to be satisfied by read and write accesses Namely, we require that each basic object X satisfy the following conditions

Semantic Conditions

1 Every CREATE(T) operation is transparent

2 For any α_1 and α_2 for which α_1CREATE(T)α_2 and $\alpha_1\alpha_2$CREATE(T) are both well-formed schedules of X, they are equieffective schedules

3 Every REQUEST_COMMIT(T,v) operation, for T a read access, is transparent

Condition (1) means that whether or not an access was created is invisible to other accesses Condition (2) means that when an access was created is not detectable by later operations Condition (3) means that later operations cannot determine whether or not a REQUEST_COMMIT operation for a read access has occurred The third condition captures the fundamental feature of read accesses that allows Moss' algorithm, as given in the construction of R/W Locking objects in Section 5 1, to work In contrast, the first two conditions are a convenience, without which the proof of correctness in Section 6 3 would be less tidy, as we would have to rearrange a concurrent schedule in more ways, to produce a serial schedule that looked the same to each transaction Note that we make no assumption about the semantics of REQUEST_COMMIT operations for write accesses, and so it is legitimate to designate all accesses as writes If this is done, Moss' algorithm as given in this paper degenerates into exclusive locking

An example of a basic object satisfying these conditions would have as its state a set of transactions, called "pending" and an instance of an abstract data type The input operation CREATE(T) would simply add T to pending At any time, a transaction T in pending could be chosen, and the corresponding function applied to the instance of the abstract data type, yielding return value v, and a possibly altered instance of the abstract type Γ would be removed from pending, the new instance would replace the old one in the state of the basic object, and REQUEST_COMMIT(T,v) would be output (The whole sequence from choosing T to the output is an atomic step of the basic object)

The following lemma combines all the information in the semantic conditions to give a simple sufficient condition for proving that schedules are equieffective This test is used throughout this paper Given a sequence α of operations of X, define write(α) to be the subsequence of α consisting of the REQUEST_COMMIT(T,v) events for write accesses T If α and β are sequences of operations of X and write(α) = write(β) then we say that α and β are *write-equal* This is clearly an equivalence relation on sequences of operations of X

Lemma 20 Let α and β be well-formed schedules of X which are write-equal Then α and β are equieffective

Proof Suppose ϕ is a sequence of operations of X such that $\alpha\phi$ and $\beta\phi$ are both well-formed We must prove that $\beta\phi$ is a schedule of X if and only if $\alpha\phi$ is a schedule of X Consider the set A of accesses to X which is the union of the set of write accesses for which a REQUEST_COMMIT operation occurs in α (and so also in β) and the set of accesses which are pending in both α and β Let α' denote the subsequence of α consisting of the events of accesses in A Similarly let β' denote the subsequence of β consisting of the events of accesses in A Since α' is obtained from α by removing all the operations of accesses not in A, and all such operations are transparent (by conditions 1 and 3), by Lemma 17, we deduce that α' is a well-formed schedule of X equieffective to α Similarly β' is a well-formed schedule equieffective to β Also, since α' can be formed from β' by moving CREATE events, we deduce from condition 2 , Lemma 16 and Lemma 15 that α' and β' are equieffective Since both $\alpha\phi$ and $\beta\phi$ are well-formed, by Lemma 3 any event in ϕ must be either an operation of an access with no operations in α or β, or else a REQUEST_COMMIT for an access that is pending in both α and β In any case, $\alpha'\phi$ and $\beta'\phi$ must be well-formed Therefore $\alpha\phi$ is a schedule of X if and only if $\alpha'\phi$ is a schedule, which is true if and only if $\beta'\phi$ is a schedule and so if and only if $\beta\phi$ is a schedule of X □

5. R/W Locking Systems

A R/W Locking system of a given system type is composed of transactions, a generic scheduler, and R/W Locking objects The non-access transactions are modelled by the same automata as in the serial system, but the generic scheduler has much more freedom in scheduling transactions than the serial scheduler, and R/W Locking objects follow the algorithm of [Mo] in maintaining locking and state restoration data that basic objects do not need

5 1 R/W Locking objects

In this section, we define, for each basic object X, a R/W Locking object M(X) which provides a resilient lock-managing variant of X It receives operation invocations and responds like basic object X, and also receives information about the fate of transactions so that it can maintain its locking and state restoration data The R/W Locking object combines the features of the resilient object and the lock manager of [LM], where, as in many database management systems, the recovery and concurrency control are performed separately Combining these features, as we do here, eliminates some redundancy in maintaining information about the fate of transactions

M(X) has the following operations

Input Operations
CREATE(T), for T an access to X
INFORM_COMMIT_AT(X)OF(T), T $\neq T_0$
INFORM_ABORT_AT(X)OF(T), T $\neq T_0$
Output Operations
REQUEST_COMMIT(T,v), for T an access to X

We give a recursive definition for *well-formedness* of schedules of object M(X) Namely, the empty schedule is well-formed Also, if $\alpha = \alpha'\pi$ is a sequence of operations of object X, then α is well-formed provided that α' is well-formed and the following hold

- If π is CREATE(T), then
  (i) there is no CREATE(T) event in α'

- If π is a REQUEST_COMMIT for T, then
  (i) there is no REQUEST_COMMIT event for T in α', and
  (ii) CREATE(T) occurs in α'

- If π is INFORM_COMMIT_AT(X)OF(T), then
  (i) there is no INFORM_ABORT_AT(X)OF(T) event in α', and
  (ii) if T is an access to X, then a REQUEST_COMMIT event for T occurs in α'

- If π is INFORM_ABORT_AT(X)OF(T), then
  (i) there is no INFORM_COMMIT_AT(X)OF(T) event in α'

A state s of M(X) consists of the following five components s.write-lockholders, s.read-lockholders, s.create_requested, and s.run, which are sets of transactions, and s.map, which is a function from write-lockholders to states of basic object X. We say that a transaction in write-lockholders *holds a write-lock*, and similarly that a transaction in read-lockholders *holds a read-lock*. We say two locks *conflict* if they are held by different transactions and at least one is a write-lock. The initial states of M(X) are those in which write-lockholders = $\{T_0\}$ and map(T_0) is an initial state of the basic object X, and the other components are empty. The transition relation of M(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
Postcondition:
  s.create_requested = s'.create_requested ∪ {T}

INFORM_COMMIT_AT(X)OF(T), T ≠ T_0
Postcondition:
  if T ∈ s'.write-lockholders then
    begin
      s.write-lockholders = (s'.write-lockholders - {T}) ∪ {parent(T)}
      s.map(U) = s'.map(U) for U ∈ s.write-lockholders - {parent(T)}
      s.map(parent(T)) = s'.map(T)
    end
  if T ∈ s'.read-lockholders then
    begin
      s.read-lockholders = (s'.read-lockholders - {T}) ∪ {parent(T)}
    end

INFORM_ABORT_AT(X)OF(T), T ≠ T_0
Postcondition:
  s.write-lockholders = s'.write-lockholders - {descendants(T)}
  s.read-lockholders = s'.read-lockholders - {descendants(T)}
  s.map(U) = s'.map(U) for all U ∈ s.write-lockholders

REQUEST_COMMIT(T,v) for T a write access to X
Precondition:
  T ∈ s'.create_requested - s'.run
  s'.write-lockholders ∪ s'.read-lockholders ⊆ ancestors(T)
  (s'.map(least(s'.write-lockholders)),CREATE(T),t)
      and (t,REQUEST_COMMIT(T,v),t')
      are in the transition relation of basic object X, for some t
Postcondition:
  s.run = s'.run ∪ {T}
  s.write-lockholders = s'.write-lockholders ∪ {T}
  s.map(U) = s'.map(U) for all U ∈ s.write-lockholders - {T}
  s.map(T) = t'

REQUEST_COMMIT(T,v) for T a read access to X
Precondition:
  T ∈ s'.create_requested - s'.run
  s'.write-lockholders ⊆ ancestors(T)
  (s'.map(least(s'.write-lockholders)),CREATE(T),t)
      and (t,REQUEST_COMMIT(T,v),t')
      are in the transition relation of basic object X, for some t
Postcondition:
  s.run = s'.run ∪ {T}
  s.read-lockholders = s'.read-lockholders ∪ {T}
```

It is clear that a R/W Locking object preserves well-formedness.

When an access transaction is created, it is added to the set create-requested. A response, containing return value v, to an access T can be returned only if the access has been requested but not yet responded to, and every holder of a conflicting lock is an ancestor of T, and v is a value which can be returned by basic object X in the response to T from some state t, obtained by performing CREATE(T) in the state map(least(write-lockholders)). When a response is given, the access transaction is added to the set run and granted the appropriate lock, and if the transaction is a write access, the resulting state is stored as map(T). If the transaction is a read access, no change is made to the stored state of basic object X, i.e. to map.

When the R/W Locking object is informed of the abort of a transaction, it removes all locks held by descendants of the transaction. When it is informed of a commit, it passes any locks held by the transaction to the parent, and also passes the version stored in map, if there is one.[8]

We introduce some terms to describe what M(X) knows about commits and aborts of transactions. If α is a sequence of operations of M(X), T is an access to X, and T' is an ancestor of T, we say that T is *committed at X* to T' in α, if α contains a subsequence β consisting of an INFORM_COMMIT_AT(X)OF(U) event for every U which is an ancestor of T and a proper descendant of T', arranged in ascending order (so the INFORM_COMMIT for parent(U) is preceded by that for U). If α is a well-formed sequence of operations of M(X) and T is an access to X and T' any transaction, we say that T is *visible at X* to T' in α if T is committed at X to lca(T,T'). We denote the subsequence of α consisting of operations of M(X) whose transactions are visible at X to T by $\text{visible}_X(\alpha,T)$. It is clear that $\text{visible}_X(\alpha,T)$ is a well-formed sequence of operations of basic object X. We say that a transaction T is an *orphan at X* in α if INFORM_ABORT_AT(X)OF(U) occurs in α for some ancestor U of T.

Here are some simple facts about the state of M(X) after a schedule α.

> **Lemma 21** Let α be a schedule of M(X), and s a state of M(X) reached by applying α to an initial state. Suppose T ∈ s.write-lockholders and T' ∈ s.read-lockholders ∪ s.write-lockholders. Then either T is an ancestor of T' or else T' is an ancestor of T.
>
> **Lemma 22** Let α be a well-formed schedule of M(X), and s a state of M(X) reached by applying α to an initial state. Let T be an access to X such that REQUEST_COMMIT(T,v) occurs in α and T is not an orphan at X in α, and let T' be the highest ancestor of T such that T is committed at X to T'. Then if T is a write access, T' must be a member of s.write-lockholders, while if T is a read access, T' must be a member of s.read-lockholders.

Given any well-formed sequence β of operations of M(X) let essence(β) denote the sequence obtained from write(β) by placing a CREATE(U) event immediately preceding a REQUEST_COMMIT(U,u) event. Since β is well-formed, essence(β) consists of a subset of the events of β and is well-formed. Clearly β and essence(β) are write-equal.

[8] If the reader wishes to compare our version of the algorithm with that in [Mo] the following may be useful. Moss gives the name "the associated state" for object X and transaction T to what we call s.map(T') where T' is the least ancestor of T in s.write-lockholders, and he calls s.map(least(s.write lockholders)) "the current state" of X. Also, he removes a read lock when the owner also holds a write lock (this is an optimization that does not affect the correctness proof). Moss also allows internal transactions to directly access objects, whereas we follow the Argus system by only having leaf transactions perform data access.

The following lemma shows how the results of operations visible at X to T are recorded in the state of M(X)

Lemma 23 Let α be a well-formed schedule of M(X) and s a state of M(X) reached by applying α to an initial state If T is a transaction which is not an orphan at X in α, then β = essence(visible$_X$(α,T)) is a well-formed schedule of X Furthermore, when β is applied to an initial state of X, it can leave X in the state s map(T') where T' is the least ancestor of T such that T' ∈ s write-lockholders

Proof By induction on the length of α The basis is trivial, so let $\alpha = \alpha'\pi$ Let s' denote a state of M(X) after applying α' such that (s',π,s) is a step of M(X) There are five cases

(1) π is CREATE(U) for an access U to X
Straightforward

(2) π is REQUEST_COMMIT(U,v) for U a read access to X
Straightforward

(3) π is REQUEST_COMMIT(U,v) for U a write access to X
We consider separately the cases U = T and U ≠ T
If U = T then T ∈ s write-lockholders so T' = T Let T'' denote the least ancestor of T in s' write-lockholders Let β' = essence(visible$_X$(α',T)) By the inductive hypothesis, β' is a well-formed schedule of X which can leave X in state s' map(T'') when applied to an initial state Now β = β'CREATE(U)π, by the definition of M(X), β'CREATE(U)π is a (well-formed) schedule of X, and applied to an initial state of X it can leave X in the state s map(T)
If U ≠ T, the proof is straightforward

(4) π is INFORM_COMMIT_AT(X)OF(U)
The discussion is divided into subcases, depending on the relation of T and U in the transaction tree

(4a) U is an ancestor of T
Now visible$_X$(α,T) = visible$_X$(α',T), so β = essence(visible$_X$(α',T)) If U is the least ancestor of T in s' write-lockholders then by the definition of M(X), T' must be parent(U) and s map(T') = s' map(U), while if U is not the least ancestor of T in s' write-lockholders then T' must be the least ancestor of T in s' write-lockholders and s map(T') = s' map(T') In either case, s map(T') is s' map(T''), where T'' is the least ancestor of T in s' write-lockholders The desired result follows immediately from the inductive hypothesis

(4b) U is not an ancestor of T, but parent(U) is an ancestor of T
Here we give separate arguments, depending on whether U is in s' write-lockholders or not If U ∈ s' write-lockholders then Lemma 21 implies that no ancestor of T that is a strict descendant of parent(U) can be in s' write-lockholders The definition of M(X) therefore shows that T' = parent(U) and that s map(T') = s' map(U) Also we note that visible$_X$(α',U) is write-equal to visible$_X$(α,T), since any write access that is committed at X to an ancestor of T in α' must be committed at X to parent(U) in α' and thus visible at X to U in α (Otherwise, by Lemma 22, some ancestor of T that is a proper descendant of parent(U) would be in s' write-lockholders)) Thus, β = essence(visible$_X$(α,T)) = essence(visible$_X$(α',U)) By the inductive hypothesis, β is a well-formed schedule of X which, when applied to an initial state of X, can leave X in state s' map(U) = s map(T')

On the other hand, if U ∉ s' write-lockholders then s write-lockholders = s' write-lockholders and s map = s' map Also, visible$_X$(α',T) is write-equal to visible$_X$(α,T) This is true because any operation visible at X to T in α is either visible at X to T in α' or else is an operation of an access that is committed at X to U in α', and any write access that is committed at X to U in α' must be committed at X to parent(U) (and hence visible at X to T) in α', by Lemma 22 and the assumption that U ∉ s' write-lockholders Thus, β = essence(visible$_X$(α,T)) = essence(visible$_X$(α',T)) By the inductive hypothesis, β is a well-formed schedule of X which, when applied to an initial state of X, can leave X in state s' map(T') = s map(T')

(4c) parent(U) is not an ancestor of T
Then visible$_X$(α,T) = visible$_X$(α',T), so β = essence(visible$_X$(α',T) Also T' is the least ancestor of T in s' write-lockholders and s' map(T') = s map(T') The desired result follows immediately from the inductive hypothesis

(5) π is INFORM_ABORT_AT(X)OF(U)
Straightforward □

A consequence of this is the following lemma, which explains a sense in which M(X) is a resilient variant of X

Lemma 24 Let α be a well-formed schedule of M(X) and T a transaction which is not an orphan at X in α Then visible$_X$(α,T) is a well-formed schedule of X

Proof We prove that visible$_X$(α,T) is a schedule of X by induction on the prefixes of visible$_X$(α,T) The base case is trivial So consider an event π in visible$_X$(α,T), and the prefix β of visible$_X$(α,T) ending with π Let $\beta = \beta'\pi$ By the inductive hypothesis β' is a well-formed schedule of X We must show that π is enabled as an operation of X after β'

If π is a CREATE event this follows from the Input Condition on all I/O automata, so suppose that π is REQUEST_COMMIT(U,u) Consider visible$_X$(γ',U) where γ = $\gamma'\pi$ is the prefix of α ending with π Let s' denote the state of M(X) immediately before π occurs, and let U' denote the least ancestor of U in s' write-lockholders By Lemma 23, β = essence(visible$_X$(γ',U)) is a well-formed schedule of X, which can leave X in state s' map(U') when applied to an initial state By the preconditions for the operation π of M(X), βCREATE(U)π is a schedule of X (Of course it is well-formed) We now show that β' and βCREATE(U) are equieffective Since each is a schedule of X, it suffices by Lemma 20 to show that they are write-equal Now βCREATE(U) and visible$_X$(γ',U) are write-equal, so we need only show that visible$_X$(γ',U) and β' are write-equal Since U is visible at X to T in α, any access visible at X to U in γ' must be visible at X to T in α, so the events in visible$_X$(γ',U) are a subset of the events in β' Now, by the preconditions for π as an operation of M(X) every element of s' write-lockholders is an ancestor of U So if REQUEST_COMMIT(V,v) occurs in β' for a write access V to X, then Lemma 22 implies that V must be committed at X to lca(V,U) in γ' (since V is not an orphan at X in γ', as it is visible at X to T in α) Thus V is visible at X to U in γ', so REQUEST_COMMIT(V,v) occurs in visible$_X$(γ',U) Also all REQUEST_COMMIT events for write accesses in visible$_X$(γ',U) occur in the same order as in α, and similarly the REQUEST_COMMIT events for write accesses in β' occur in the same order as in α Thus $\text{visible}_X(\gamma',U)$ and β' are write-equal, completing the proof that βCREATE(U) and β' are equieffective Since βCREATE(U)π is a well-formed schedule of X and $\beta = \beta'\pi$ is well-formed, the definition of equieffective implies that β is a well-formed schedule of X, as required Thus, by induction, $\text{visible}_X(\alpha,T)$ is a well-formed schedule of X □

5 2 Generic Scheduler

The generic scheduler is a very nondeterministic automaton It passes requests for the creation of sub-transactions or accesses to the appropriate recipient, passes responses back to the caller and informs objects of the fate of transactions, but it may delay such messages for arbitrary lengths of time or unilaterally decide to abort a subtransaction which has been created Moss [Mo] devotes considerable effort to describing a distributed implementation of the scheduler that copes with communication failures and loss of system information due to crashes, yet still commits a subtransaction whenever possible These concerns are orthogonal to the correctness of the data management algorithms and we do not address them here [9]

The generic scheduler has nine operations

Input Operations
    REQUEST_CREATE(T)
    REQUEST_COMMIT(T,v)
Output Operations
    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), $T \neq T_0$

These play the same roles as in the serial scheduler, except for the INFORM_COMMIT and INFORM_ABORT operations which pass information about the fate of transactions to the R/W Locking objects

Each state s of the generic scheduler consists of six sets s create_requested, s created, s commit_requested, s committed, s aborted and s returned The set s commit_requested is a set of (transaction,value) pairs, and the others are sets of transactions All are empty in the initial state except for create_requested, which is $\{T_0\}$

The operations are defined by pre- and postconditions as follows

REQUEST_CREATE(T)
Postcondition
    s create_requested = s' create_requested ∪ {T}

REQUEST_COMMIT(T,v)
Postcondition
    s commit_requested = s' commit_requested ∪ {(T,v)}

CREATE(T), T a transaction
Precondition
    T ∈ s' create_requested - s' created
Postcondition
    s created = s' created ∪ {T}

COMMIT(T), $T \neq T_0$
Precondition
    (T,v) ∈ s' commit_requested for some v
    T ∉ s' returned
    children(T) ∩ s' create_requested ⊆ s' returned
Postcondition
    s committed = s' committed ∪ {T}
    s returned = s' returned ∪ {T}

ABORT(T), $T \neq T_0$
Precondition
    T ∈ s' create-requested - s' returned
Postcondition
    s aborted = s' aborted ∪ {T}
    s returned = s' returned ∪ {T}

REPORT_COMMIT(T,v), $T \neq T_0$
Precondition
    T ∈ s' committed
    (T,v) ∈ s' commit_requested

REPORT_ABORT(T), $T \neq T_0$
Precondition
    T ∈ s' aborted

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 25 Let α be a schedule of the generic scheduler, and let s be a state which can result from applying α to the initial state s_0 Then the following conditions are true

1 T is in s create_requested exactly if α contains a REQUEST_CREATE(T) event

2 T is in s created exactly if α contains a CREATE(T) event

3 (T,v) is in s commit_requested exactly if α contains a REQUEST_COMMIT(T,v) event

4 T is in s committed exactly if α contains a COMMIT(T) event

5 T is in s aborted exactly if α contains an ABORT(T) event

6 s returned = s committed ∪ s aborted

7 s committed ∩ s aborted = ∅

5 3 R/W Locking Systems

The composition of transactions with R/W Locking objects and the generic scheduler is called a *R/W Locking system*, and its operations and schedules are called *concurrent operations* and *concurrent schedules*, respectively [10] A sequence α of concurrent operations is said to be *well-formed* provided that its projection at every transaction and R/W Locking object is well-formed

[9] The generic scheduler is very similar to the weak concurrent controller of [LM] It differs slightly in the names of its operations, in the separation of return and report operations, and in the conditions under which CREATE operations are permitted to occur

[10] Note that this usage differs from that in [LM]

Lemma 26. If α is a concurrent schedule, then α is well-formed

The following lemma is straightforward

Lemma 27 Let α be a concurrent schedule If T is a transaction that is not an orphan in α and T' is visible to T in α, then T' is not an orphan in α

Note that if α is a concurrent schedule then any INFORM_COMMIT_AT(X)OF(T) is preceded by a COMMIT(T) event (by the scheduler preconditions) and similarly any INFORM_ABORT_AT(X)OF(T) is preceded by ABORT(T) Thus, if T is visible at X to T' in α then T is visible to T' in α, and if T is an orphan at X in α then T is an orphan in α Thus, $\text{visible}_X(\alpha,T)$ is a subsequence of visible(α,T)|X when α is a concurrent schedule

A key property of R/W Locking systems is given next

Lemma 28 Let α be a concurrent schedule, T a transaction that is not an orphan in α and M(X) a R/W Locking object Then visible(α,T)|X is a schedule of basic object X

Proof Let S denote the set of transactions with COMMIT events in α Construct a sequence β by appending to α a sequence of INFORM_COMMIT_AT(X)OF(U) events, where the U give a post-order traversal of S Since α contains a COMMIT(U) event for each U in S, β is a concurrent schedule, and by Lemma 24 $\text{visible}_X(\beta,T)$ is a schedule of X Since the INFORM_COMMIT_AT(X)OF(U) events at the end of β are in ascending order, and occur for every U that is visible to T in β, $\text{visible}_X(\beta,T)$ = visible(β,T)|X Also visible(β,T) = visible(α,T) since INFORM_COMMIT operations have no influence on what transactions are visible to T Thus visible(α,T)|X is a schedule of X □

6. The Proof of Serial Correctness

We prove that a R/W Locking system generates schedules that are serially correct for each non-orphan transaction T, by taking a concurrent schedule α, extracting the subsequence visible(α,T) of events whose effects might have been detected by T, and then rearranging the operations in this to give a serial schedule β The rearrangements permitted are those that transform one sequence into a "write-equivalent" one

6 1 Write-Equivalence

Two sequences of serial operations, α and β, are *write-equivalent* if

1 they contain the same events,

2 for each transaction U, α|U = β|U, and

3 for each basic object X, α|X and β|X are write-equal sequences of operations of X

Thus, the rearrangements allowed include interchanging the order of two events of different transactions or objects, and also interchanging the order of events of a single object, provided that they are not both REQUEST_COMMITs for write accesses By the semantic conditions of Section 4 3, such rearrangements at objects are such that the difference between the orders is not detectable by any later operations of that object This property is expressed by the following lemma

Lemma 29 If α and β are write equivalent sequences, and α/X and β/X are well formed schedules of X, then for each basic object X, α|X and β|X are equieffective sequences

Write-equivalence is obviously an equivalence relation We have some straightforward results

Lemma 30 If α and β are well-formed sequences of operations which are write-equivalent, then $\beta\phi$ is write-equivalent to $\alpha\phi$

Lemma 31 If α and β are serial schedules which are write-equivalent and $\alpha\phi$ is a serial schedule then $\beta\phi$ is a serial schedule

6 2 A Technical Lemma

In this subsection, we prove an extension of Lemma 14, for use in the proof of Lemma 33, in Section 6 3

Lemma 32 Let α be a concurrent schedule, and let T and T' be two non-orphan transactions with T' visible to T in α Let β and β_1 be serial schedules, such that β is write-equivalent to visible(α,T) and β_1 is write-equivalent to visible(α,T') Then $\gamma = \beta_1(\beta - \beta_1)$ is a serial schedule which is write-equivalent to visible(α,T)

Proof First we prove that β' = visible(β,T') is write-equivalent to β_1 By Lemma 10 and Lemma 8, β' and β_1 contain the same events For any basic object X, write(β'|X) = write(β_1|X) since REQUEST_COMMIT events for write accesses to X occur in β' in the same order as they occur in β, which is the same as the order they occur in α, which is the same as the order they occur in β_1 For any transaction U which is visible to T' in α (and hence in β), β'|U = β|U = α|U, by Lemma 9 and write-equivalence, and similarly β_1|U = α|U On the other hand, if U is not visible to T' in α, β'|U and β_1|U are both empty For later use we note that β' is a serial schedule, by Lemma 13, and β_1 is a serial schedule, so β'|X and β_1|X are schedules of X

By Lemma 14, $\beta'(\beta - \beta')$ is a serial schedule Since $\beta - \beta'$ = $\beta - \beta_1$ (as β' and β_1 contain the same events) we deduce from Lemma 31 and the fact that β' and β_1 are write-equivalent, that γ is a serial schedule

Next, we prove that write(visible(α,T')|X) is a prefix of write(visible(α,T)|X) for any object X So suppose that visible(α,T) contains a REQUEST_COMMIT(U,u) event for a write access U to X which is not in visible(α,T') Let REQUEST_COMMIT(U',u') be a subsequent event, where U' is a write access to X which is visible to T in α We must show that U' is not visible to T' in α Consider the prefix δ of α which precedes the REQUEST_COMMIT(U',u'), and let s denote the state of the R/W Locking object M(X) after δ If we denote by U'' the highest ancestor of U to which U has committed in α, then U'' is a proper descendant of lca(U,T'), since U is not visible to T' in α Then the highest ancestor of U to which U is committed at X in δ must be a descendant of U'', and so by Lemma 22 some descendant of U'' is in s write-lockholders By the preconditions for the operation REQUEST_COMMIT(U',u') of M(X), U' must be a descendant of U'', and therefore U' is not committed in α to lca(U',T') = lca(U'',T') = lca(U,T') Therefore U' is not visible to T' in α, establishing that write(visible(α,T')|X) is a prefix of write(visible(α,T)|X)

Now we show that γ is write-equivalent to β They clearly contain the same events, since every event of β_1 occurs in β (because any operation visible to T' in α is also visible to T in α by Lemma 7) If P is a basic object, write(β_1|P) =

write(visible(α,T'$)|$P) is a prefix of write(visible(α,T$)|$P) = write($\beta|$P), so that write($\gamma|$P) = (write($\beta_1|$P))(write($\beta|$P) - write($\beta_1|$P)) = write($\beta|$P) If P is a transaction that is visible to T' in α then $\beta_1|$P = visible(α,T'$)|$P = $\alpha|$P = visible(α,T$)|$P = $\beta|$P, so $\gamma|$P = ($\beta_1|$P)($\beta|$P - $\beta_1|$P) = $\beta|$P

On the other hand, if P is a transaction not visible to T' in α then $\beta_1|$P is empty, so trivially $\gamma|$P = $\beta|$P

Since γ is write-equivalent to β, it is write-equivalent to visible(α,T) □

6 3 The Main Results

We are now ready to prove that R/W Locking systems are serially correct for every transaction that is not an orphan We actually state a stronger property, which carries useful invariants through the induction

Lemma 33 Let α be a concurrent schedule, and T any transaction which is not an orphan in α Then there is a serial schedule β which is write-equivalent to visible(α,T)

Proof The proof follows the outlines of that of the main theorem of [LM] We proceed by induction on the length of α As before, let $\alpha = \alpha'\pi$ We must show that there is a serial schedule β which is write-equivalent to visible(α,T) We can assume that transaction(π) is visible to T in α There are seven cases, and in each we relate visible(α,T) to visible(α',U) for one or more transactions U, and build β from serial schedules write-equivalent to visible(α',U)

(1) π is an output operation of a non-access transaction T'
Since T is not an orphan in α', the inductive hypothesis implies the existence of a serial schedule β' which is write-equivalent to visible(α',T) Let $\beta = \beta'\pi$ We will show that β is a serial schedule that is write-equivalent to visible(α,T) By Lemma 1, to check that β is a serial schedule we need only check that $\beta'\pi|$T' is a schedule of T' However $\beta'|$T' = visible(α',T$)|$T' = $\alpha'|$T' by Lemma 9 (since T' is visible to T) Thus $\beta'\pi|$T' = $\alpha'\pi|$T' = $\alpha|$T' which is a schedule of T' Thus, β is a serial schedule

By Lemma 11, visible(α,T) = visible(α',T)π and since β' is write-equivalent to visible(α',T), we may apply Lemma 30 to deduce that β is write-equivalent to visible(α,T)

(2) π is an output operation of an access T' to a R/W Locking object M(X)
Define β' and β as in the previous case As before, to check that β is a serial schedule, we need only check that $\beta'\pi|$X is a schedule of X However, Lemma 29 implies that $\beta'|$X is equieffective to and contains the same events as visible(α',T$)|$X Now visible(α',T)$\pi|$X = visible($\alpha'\pi$,T$)|$X = visible(α,T$)|$X which is a schedule of X by Lemma 28 Thus by Lemma 16, $\beta'\pi|$X is a schedule of X Thus, β is a serial schedule

Since visible(α,T) = visible(α',T)π, $\beta = \beta'\pi$, and β' is write-equivalent to visible(α',T), we may apply Lemma 30 to deduce that β is write-equivalent to visible(α,T)

(3) π is a CREATE(T') operation
Then transaction(π) = T', and so T' is visible to T in α By well-formedness and the scheduler preconditions, any operation of a proper descendant of T' must be preceded by a REQUEST_CREATE for a child of T', and by well-formedness any operation of T' must follow CREATE(T') Thus, π is the first event whose transaction is a descendant of T', so T' = T Now, parent(T) is not an orphan in α, and hence in α', so the inductive hypothesis implies the existence of a serial schedule β' which is write-equivalent to visible(α',parent(T)) Let $\beta = \beta'\pi$ We will show that β is a serial schedule that is write-equivalent to visible(α,T)

To show that β is a serial schedule, we need only check that $\beta'\pi$ is a schedule of the serial scheduler Let s' be the state of the serial scheduler after β', and s'' the state of the generic scheduler after α' Since π is enabled in s'', Lemmas 25 and 4 and the preconditions imply that π is enabled in s', hence $\beta'\pi$ is a schedule of the serial scheduler, and β is serial schedule

Since visible(α,T) = visible(α',parent(T))π, $\beta = \beta'\pi$ and β' is write-equivalent to visible(α',parent(T)), we may deduce from Lemma 30 that β is write-equivalent to visible(α,T)

(4) π is a COMMIT(T') operation
Then T'' = parent(T') is visible to T in α, since transaction(π) = T'' Then COMMIT(T') does not occur in α', and so T must be a descendant of T'' (since T'' is visible to T) Also, by Lemma 27, T'' is not an orphan in α and so also T'' is not an orphan in α' From this, we deduce that T' is not an orphan in α' We distinguish two cases, depending on whether T is a descendant of T' or not

If T is a descendant of T', the argument is straightforward If T is not a descendant of T', the inductive hypothesis yields three serial schedules, β', β'' and γ, which are write-equivalent to visible(α',T'), visible(α',T) and visible(α',T'') respectively Let $\beta_1 = \beta' - \gamma$ and $\beta_2 = \beta'' - \gamma$ Let $\beta = \gamma\beta_1\pi\beta_2$ We show that β is a serial schedule that is write-equivalent to visible(α,T) That β is serial follows from Lemma 18, provided we can show that

(4 a) $\gamma\beta_1\pi$ is a serial schedule,

(4 b) $\gamma\beta_2$ is a serial schedule,

(4 c) $\gamma\beta_1$ = visible($\gamma\beta_1$,T'),

(4 d) $\gamma\beta_2$ = visible($\gamma\beta_2$,T),

(4 e) γ = visible($\gamma\beta_1$,T'') = visible($\gamma\beta_2$,T'') and

(4 f) if any basic object X has an output operation in β_2 then every operation in $\beta_1|$X is transparent

(4 a) By Lemma 32, $\gamma\beta_1$ is a serial schedule (and is write-equivalent to visible(α',T')) We must therefore show that π is enabled at the serial scheduler after $\gamma\beta_1$ The serial scheduler preconditions and Lemma 4 show that we must prove that REQUEST_COMMIT(T',v) occurs in $\gamma\beta_1$ for some v, that no return for T' occurs in $\gamma\beta_1$, and that for every child U of T' with a REQUEST_CREATE(U) in $\gamma\beta_1$ there is a return event in $\gamma\beta_1$ Since π is enabled in the generic scheduler after α', each of these is true with α' replacing $\gamma\beta_1$ Since all these operations are visible to T' in α', all these statements are also true of visible(α',T') and thus of the write-equivalent sequence $\gamma\beta_1$, as required Thus $\gamma\beta_1\pi$ is a serial schedule We also note that Lemma 30 proves that $\gamma\beta_1\pi$ is write-equivalent to visible(α,T') = visible(α',T')π

(4 b) By Lemma 32, $\gamma\beta_2$ is serial (and write-equivalent to visible(α',T))

Parts (4 c)-(4 e) are immediate

(4 f) We prove that if a basic object X has an output operation in β_2 then no event in $\beta_1|X$ is a REQUEST_COMMIT for a write access Suppose this were false Then β_1 contains a REQUEST_COMMIT(V_1,v_1) for V_1 a write access to X, and β_2 contains a REQUEST_COMMIT(V_2,v_2) for V_2 an access to X Since V_1 is visible in α to T' but not to T", V_1 must be a descendant of T', and not an orphan in α, and V_1 must not be committed at X to T" in α By Lemma 22, some descendant of T' is in s write-lockholders, where s is a state of M(X) after applying α Similarly V_2 must be a descendant of some sibling U of T' but not committed at X to T" in α, so by Lemma 22, some descendant of U is in s readlockholders $\cup$ s write-lockholders But these two statements about lockholders contradict Lemma 21

Now we must prove that β is write-equivalent to visible(α,T) Since any transaction visible to T in α is either visible to T in α' or visible to T' in α' and if both then it is visible to T" in α', it is clear that β and visible(α,T) contain the same events If P is a basic object, either β_2 contains no output operations of P or else no operation in $\beta_1|P$ is a REQUEST_COMMIT for a write access In the first case write($\beta_2|P$) is empty, and since write($\gamma\beta_1\pi|P$) = write(visible(α',T')|P), we have write($\beta|P$) = write(visible(α,T)|P) In the second case write($\beta_1|P$) is empty, and since write($\gamma\beta_2|P$) = write(visible(α',T)|P), we again have write($\beta|P$) = write(visible(α,T)|P) If P is a non-access transaction which is not visible to T in α, then no operations occur at P in either β or visible(α,T) For P any non-access transaction which is visible to T in α, either P is visible to T in α' or P is visible to T' in α' In the first case, $\beta_2|P$ is empty so $\beta|P = \gamma\beta_1\pi|P$ = visible(α,T')|P as we saw above that $\gamma\beta_1\pi$ and visible(α,T') are write-equivalent, and visible(α,T')|P = α|P = visible(α,T)|P Similarly in the second case $\beta_1\pi|P$ is empty and $\beta|P = \gamma\beta_2|P$ = visible(α',T)|P = visible(α,T)|P In every case, we have checked that $\beta|P$ = visible(α,T)|P Thus β and visible(α,T) are write-equivalent

(5) π is an ABORT(T') operation
Then T" = parent(T') is visible to T in α, since π has transaction T" Then COMMIT(T") does not occur in α' and so T must be a descendant of T" (since T" is visible to T) Also by Lemma 27, T" is not an orphan in α and so also T" is not an orphan in α' Since T is not an orphan in α, T is not a descendant of T' Thus the inductive hypothesis yields two serial schedules, β' and γ, which are write-equivalent to visible(α',T) and visible(α',T") respectively Let $\beta_1 = \beta' - \gamma$ Let $\beta = \gamma\pi\beta_1$ We show that β is a serial schedule that is write-equivalent to visible(α,T) That β is serial follows from Lemma 19, provided we can show that

(5 a) $\gamma\pi$ is a serial schedule,

(5 b) $\gamma\beta_1$ is a serial schedule,

(5 c) $\gamma\beta_1$ = visible($\gamma\beta_1$,T),

(5 d) γ = visible(γ,T") = visible($\gamma\beta_1$,T")

(5 a) Since γ is a serial schedule, we must show that π is enabled at the serial scheduler after γ The serial scheduler preconditions and Lemma 4 show that we must prove that REQUEST_CREATE(T') occurs in γ, and that no CREATE(T') or ABORT(T') occurs in γ Since π is enabled in the generic scheduler after α', α' contains a REQUEST_CREATE(T') event, and since this operation has transaction T", REQUEST_CREATE(T') is in visible(α',T") and hence in γ Thus, T' is not committed in α', so that any CREATE(T') event in α' is not visible to T", and so does not occur in visible(α',T") and hence does not occur in γ There is no ABORT(T') event in α', so ABORT(T') does not occur in γ Thus $\gamma\pi$ is a serial schedule We also note that Lemma 30 proves that $\gamma\pi$ is write-equivalent to visible(α,T') = visible(α',T')π, since γ and visible(α',T') are write-equivalent

(5 b) By Lemma 32, $\gamma\beta_1$ is a serial schedule (and it is write-equivalent to visible(α',T))

Parts (5 c) and (5 d) are immediate

Now we must prove that β is write-equivalent to visible(α,T) Since any transaction visible to T in α is visible to T in α', and either visible to T" in α' or not, it is clear that β and visible(α,T) contain the same events If P is a basic object, since write($\gamma\beta_1|P$) = write(visible(α',T')|P) we have write($\beta|P$) = write(visible(α,T)|P) For P any non-access transaction, $\beta|P = \gamma\beta_1|P$ = visible(α',T)|P = visible(α,T)|P, since $\pi|P$ is empty and $\gamma\beta_1$ and visible(α',T) are write-equivalent This completes the demonstration that β and visible(α,T) are write-equivalent

(6) π is REPORT_COMMIT(T',v)
Since T is not an orphan in α' there is a serial schedule β' which is write-equivalent to visible(α',T) Put $\beta = \beta'\pi$ By the preconditions of the generic scheduler and Lemma 25, REQUEST_COMMIT(T',v) and COMMIT(T',v) occur in α' Since the report is in visible(α',T), parent(T') is visible to T in α', thus, COMMIT(T',v), and hence REQUEST_COMMIT(T',v), are in visible(α',T) So COMMIT(T',v) and REQUEST_COMMIT(T',v) occur in β' The serial scheduler preconditions and Lemma 4 imply that π is enabled after β' at the serial scheduler, and so by Lemma 1 and Lemma 30, β is a serial schedule that is write-equivalent to visible(α,T) = visible(α',T)π

(7) π is REPORT_ABORT(T')
This is just like case (6)

Thus in every case we have produced a serial schedule β that is write-equivalent to visible(α,T) □

Theorem 34· Every concurrent schedule is serially correct for every non-orphan non-access transaction

Proof Let T be a transaction that is not an orphan in the concurrent schedule α By Lemma 33 there is a serial schedule β that is write-equivalent to visible(α,T) Then α|T = visible(α,T)|T by Lemma 9, and by write-equivalence, visible(α,T)|T = β|T □

Corollary 35 Every concurrent schedule is serially correct for T_0

7. Acknowledgements

We thank the members of the Theory of Distributed Systems seminar at MIT for many helpful suggestions

8 References

[A] Allchin, J E , "An Architecture for Reliable Decentralized Systems", Ph D Thesis, School of Info and Comp Sci , Georgia Institute of Technology, September 1983

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

[BBGLS] Beeri, C , Bernstein, P A , Goodman, N , Lai, M Y , and Shasha, D E , "A Concurrency Control Theory for Nested Transactions," *Proc 1983 Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August 17-19, 1983, pp 45-62

[BG] Bernstein, P A , and Goodman, N , "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* 13,2 (June 1981), pp 185-221

[D] Davies, C T , "Recovery Semantics for a DB/DC System," *Proc ACM National Conference 28*, 1973, pp 136-141

[EGLT] Eswaren, K P , Gray, J N , Lorie, R A , and Traiger, I L , "The Notions of Consistency and Predicate Locks in Database Systems," *Communications of the ACM*, Vol 19, No 11, November 1976, pp 624-633

[Go] Goree, J , "Internal Consistency Of A Distributed Transaction System With Orphan Detection," MS Thesis, TR-286, Laboratory for Computer Science, MIT, January 1983

[Gr] Gray, J , "Notes on Database Operating Systems," in Bayer, R , Graham, R and Seegmuller, G (eds), Operating Systems an Advanced Course, Lecture Notes in Computer Science, Vol 60, Springer-Verlag, 1978

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

[Ho] Hoare, C A R , "Communicating Sequential Processes," Prentice Hall International, 1985

[Ko] Korth, H F , "Deadlock Freedom Using Edge Locks," *ACM Trans on Database Systems*, Vol 7, No 4, December 1982, pp 632-652

[KS] Kedem, Z , and Silberschatz, A , "Non-two phase locking protocols with shared and exclusive locks," *Proc Int Conference on Very Large Data Bases*, 1980, pp 309-320

[LHJLSW] Liskov, B , Herlihy, M , Johnson, P , Leavens, G , Scheifler, R , and Weihl, W , "Preliminary Argus Reference Manual," Programming Methodology Group Memo 39, October 1983

[LiS] Liskov, B , and Scheifler, R , "Guardians and Actions Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems* Vol 5, No 3, July 1983, pp 381-404

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

[LT] Lynch, N , and Tuttle, M , "Correctness Proofs for Distributed Algorithms," in progress

[Ly] Lynch, N A , "Concurrency Control For Resilient Nested Transactions," *Advances in Computing Research* 3, 1986, pp 335-373

[MGG] Moss, J E B , Griffeth, N D , and Graham, M H , "Abstraction in Concurrency Control and Recovery Management" Technical Report 86-20, COINS University of Massachussetts, Amherst, MA , May 1986

[Mo] Moss, J E B , "Nested Transactions An Approach To Reliable Distributed Computing," Ph D Thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA , April 1981 Also, published by MIT Press, March 1985

[P] Papadimitriou, C H , "The Serializability of Concurrent Database Updates," *J ACM* Vol 26, No 4, October 1979, pp 631-653

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

[T] Thomas, R H , "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans on Database Systems*, Vol 4, No 2, June 1979, pp 180-209

[We] Weihl, W E , "Specification and Implementation of Atomic Data Types," Ph D Thesis, Technical Report/MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, MA , March 1984