Introduction to the Theory of Nested Transactions

Nancy Lynch
Massachusetts Institute of Technology
Cambridge, Mass.

Michael Merritt
A. T. and T. Bell Laboratories
Murray Hill, New Jersey

July 21, 1986

1. Introduction

This paper develops the foundation for a general theory of nested transactions. We present a simple formal model for studying concurrency and resiliency in a nested environment. This model has distinct advantages over the many alternatives, the greatest of which is the unification of a subject replete with formalisms, correctness conditions and proof techniques. The authors are presently engaged in an ambitious project to recast the substantial amount of work in nested transactions within this single intuitive framework. These pages contain the preliminary results of that project – a description of the model, and its use in stating and proving correctness conditions for two variations of a well-known algorithm.

The model is based on *I/O automata*, a simple formalization of communicating automata. It is not complex – it is easily presented in a few pages, and easy to understand, given a minimal background in automata theory. Each nested transaction and data object is modelled by a separate I/O automaton. These automata, the system *primitives*, issue requests to and receive replies from some *scheduler*, which is simply another I/O automaton. Simple syntactic constraints on the interactions of these automata ensure, for example, that no transaction requests the creation of the same child more than once. One scheduler, in this case the "serial scheduler", interacts with the transactions and objects in a particularly constrained way. The "serial schedules" of the primitives and the serial scheduler are the basis of our correctness conditions. Specifically, alternative schedulers are required to ensure that nested transaction automata individually have local schedules which they could have in a serial schedule. In essence, each scheduler must "fool" the transactions into believing that the system is executing in conjunction with the serial scheduler.

In the past ten years, an important and substantial body of work has appeared on the design and analysis of algorithms for implementing concurrency control and resiliency in database transaction systems [EGLT,RLS,BG,KS,Gr,LaS, etc.]. Among this has been a number of results dealing with nested transactions [R,Mo,LiS,LHJLSW,AM,BBGLS,BBG, etc.]. The present work does not replace these other contributions, but augments them by providing a unifying and mathematically tractable framework for posing and exploring a variety of questions. This previous work uses behavioral specifications of nested transactions, focusing on what nested transactions do, rather than what they are. By answering the question "What is a nested transaction?", I/O automata provide a powerful tool for understanding and reasoning about them.

Some unification is vitally important to further development in this field. The plethora and complexity of existing formalizations is a challenge to the most seasoned researcher. More critically, it belies the argument that nested transactions provide a clean and intuitive tool for organizing distributed databases and more general distributed applications. It is particularly important to provide an intuitive and precise description of nested transactions themselves, as in typical systems, these are the components which the application programmer must implement!

The remainder of this paper is organized as follows. The I/O automaton model is described in Section 2. The rest of the paper contains an extended example, which establishes correctness properties for two related lock-based concurrent schedulers.

Section 3 contains simple definitions for naming nested transactions and objects, and for specifying the operations (interactions) of these components. Simple syntactic restrictions on the orders of these operations are presented, and then a particular system of I/O automata is presented, describing the interactions of nested transactions and objects with a serial scheduler. The interface between the serial scheduler and the transactions provides a basis for the specification of correctness conditions for alternative schedulers. These schedulers would presumably be more efficient than the serial scheduler. The strongest correctness condition, "serial correctness," requires that *all* non-access transactions see serial behavior at their interface with the system. The second condition, "correctness for T_0," only requires that this serial interface be maintained at the interface of the system and the external world. These interfaces also provide simple descriptions of the environment in which nested transactions can be assumed to execute. A particular contribution is the clear and concise semantics of ABORT operations which arises naturally from this formalization. The section closes with some lemmas describing useful properties of serial systems.

Next, a lock-based concurrent system is presented. Section 4 contains a description of a special type of object, called a "resilient object", which is used in the concurrent system. Section 5 describes the remainder of the concurrent system, the "concurrent scheduler." This concurrent scheduler includes "lock manager" modules for all the objects; lock managers coordinate concurrent accesses.

Section 6 defines a system which is closely related to the concurrent system, the "weak concurrent system." This system preserves serial correctness for those transactions whose ancestors do not abort (i.e.. those that are not "orphans"). Since the root of the transaction tree, T_0, has no ancestor, weak concurrent systems are correct for T_0. Section 7 contains correctness theorems for concurrent and weak concurrent sytems; concurrent systems are serially correct, and weak concurrent systems are correct for T_0. The stronger condition is obtained for concurrent systems as a corollary to a result about weak concurrent systems.

It is interesting that the concurrent system algorithms are described in complete detail (essentially, in "pseudocode"), yet significant formal claims about their behavior can be stated clearly and easily. Despite the detailed level of presentation, the underlying model is general enough that the results apply to a wide range of implementations.

The style of the correctness proof is also noteworthy. It is a constructive proof, in that for each step of the weak concurrent system and each non-orphan transaction, an execution of the serial system is explicitly constructed. The transaction's local "view" in the constructed execution is identical to that in the original weak concurrent execution, establishing the correctness of the weak concurrent system. One may think of the weak concurrent system as maintaining consistent, parallel "world views" within which concurrent siblings execute. As siblings return to their parent, these parallel worlds are "merged" to form a single consistent view. The locking policy prevents collisions between different views at the shared data. This intuition is strongly supported and clarified by the correctness proof, which constructs the parallel views as different serial schedules consistent with each sibling's local history. Lemmas illustrate how these serial schedules can be merged as siblings return or abort to their parent.

Section 8 contains a discussion of the relationship of this work to previous results, and Section 9 contains an indication of the work that lies ahead.

This paper is a shortened version of a complete paper with the same title, available as a technical report from MIT or AT&T Bell Labs [LM]. Whereas the present paper omits many easy lemmas and straightforward proofs, the longer paper contains completely detailed proofs of all results.

2. Basic Model

In this section, we present the basic I/O automaton model, which is used to describe all components of our systems. This model consists of rather standard, possibly infinite-state, nondeterministic automata that have operation names associated with their state transitions. Communication among automata is described by identifying their operations. This model is very similar to models used by Milner, Hoare [Mi,Ho] and others. There are a few differences: first, we find it important to classify operations of any automaton or system of automata as either "input" or "output" operations, of that automaton or system, and we treat these two cases differently. Also, we allow identification of arbitrary numbers of operations from different automata, rather than just pairwise identification as considered in [Mi].

This paper is not intended to develop the basic model. For the general theory of I/O automata, including a unified treatment of finite and infinite behavior, we refer the reader to [LT]. In the present treatment of concurrent transaction systems, we only prove properties of finite behavior, so we only require a simple special case of the general model.

2.1. I/O Automata

All components in our systems, transactions, objects and schedulers, will be modelled by *I/O automata*. An I/O automaton $\mathcal{A}$ has components *states*($\mathcal{A}$), *start*($\mathcal{A}$), *out*($\mathcal{A}$), *in*($\mathcal{A}$), and *steps*($\mathcal{A}$). Here, *states*($\mathcal{A}$) is a set of states, of which a subset *start*($\mathcal{A}$) is designated as the set of start states. The next two components are disjoint sets: *out*($\mathcal{A}$) is the set of *output operations*, and *in*($\mathcal{A}$) is the set of *input operations*. The union of these two sets is the set of *operations* of the automaton. Finally, *steps*($\mathcal{A}$) is the transition relation of $\mathcal{A}$, 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 $\mathcal{A}$.

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. Our partitioning of operations into input and output indicates that each operation is only triggered in one place. We require the following condition.

Input Condition: For each input operation π and each state s', there exist a state s and a step (s',π,s).

This condition says that an I/O automaton must be prepared to receive any input operation at any time. This condition makes intuitive sense if we think of the input operations as being triggered externally. (In this paper, this condition serves mainly as a technical convenience, but in [LT], where infinite behavior is considered, it is critical.)

An *execution* of $\mathcal{A}$ is an alternating sequence $s_0,\pi_1, s_1,\pi_2,\ldots$ of states and operations of $\mathcal{A}$; the sequence may be infinite, but if it is finite, it ends with a state. Furthermore, s_0 is in start($\mathcal{A}$), 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.

If S is any set of schedules (or property of schedules), then $\mathcal{A}$ is said to *preserve* S provided that the following holds. If $\alpha = \alpha'\pi$ is any schedule of $\mathcal{A}$, where π is an output operation, and α' is in S, then α is in S. That is, the automaton is not the first to violate the property described by S.

2.2. Composition of Automata

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* $\mathcal{S}$, if all of the output operations are disjoint. (Thus, every output operation in $\mathcal{S}$ will be triggered by exactly one component.) The system $\mathcal{S}$ is itself an I/O automaton. 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 set of *operations* of $\mathcal{S}$, *ops*($\mathcal{S}$), is exactly the union of the sets of operations of the component automata. The set of *output operations* of $\mathcal{S}$, *out*($\mathcal{S}$), is likewise the union of the sets of output operations of the component automata. Finally, the set of *input operations* of $\mathcal{S}$, *in*($\mathcal{S}$), is *ops*($\mathcal{S}$) - *out*($\mathcal{S}$), the set of operations of $\mathcal{S}$ that are not output operations of $\mathcal{S}$. 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.

The triple (s',π,s) is in the transition relation of $\mathcal{S}$ if and only if for each component automaton $\mathcal{A}$, one of the following two conditions holds. Either π is an operation of $\mathcal{A}$, and the projection of the step onto $\mathcal{A}$ is a step of $\mathcal{A}$, or else π is not an operation of $\mathcal{A}$, and the states corresponding to $\mathcal{A}$ in the two tuples s' and s are identical. Thus, each operation of the composed automaton is an operation of a subset of the component automata. During an operation π of $\mathcal{S}$, each of the components which has operation π carries out the operation, while the remainder stay in the same state. Again, the operation π is an output operation of the composition if it is the output operation of a component – otherwise, π is an input operation of the composition.[1] An *execution* of a system is defined to be an execution 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 $\mathcal{A}$.

Lemma 1: Let α' be a schedule of a system $\mathcal{S}$, 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 $\mathcal{S}$.

[1] Note that our model has chosen a particular convention for identifying operations of different components in a system: we simply identify those with the same name. This convention is simple, and sufficient for what we do in this paper. However, when this work is extended to more complicated systems, it may be expedient to generalize the convention for identifying operations, to permit reuse of the same operation name internally to different components. This will require introducing a renaming operator for operations, or else defining composition with respect to a designated equivalence relation on operations. We leave this for later work.

3. Serial Systems

In this paper, we define three kinds of systems: "serial systems" and two types of "concurrent systems". Serial systems describe serial execution of transactions. Serial systems are defined for the purpose of providing a correctness condition for other systems: that the schedules of the other systems should "look like" schedules of the serial system to the transactions. As with serial schedules of single-level transaction systems, our serial schedules are too inefficient to use in practice. Thus, we define systems which allow concurrency, and which permit the abort of transactions after they have performed some work. We then prove that the schedules permitted by concurrent systems are correct.

In this section, we define "serial systems". Serial systems 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 defines the allowable orders in which the transactions may take steps. All three types of system components are modelled as I/O automata.

We begin by defining a structure which describes the nesting of transactions. Namely, a *system type* is a four-tuple ($\mathcal{T}$,parent,$\mathcal{O}$,V), where $\mathcal{T}$, the set of transaction names, is organized into a tree by the mapping parent:$\mathcal{T} \rightarrow \mathcal{T}$, 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 set $\mathcal{O}$ denotes the set of objects; formally, $\mathcal{O}$ is a partition of the set of accesses, where each element of the partition contains the accesses to a particular object. The set V is a set of *values*, to be used as return values of transactions.

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.

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 [LiS,LHJLSW], 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 much the same way as about all of the other transactions, although it is distinguished from the other transactions by having no parent transaction. Since committing and aborting are operations which take place at the parent of each transaction (see below), T_0 can neither commit nor abort.

Thus, a commit or abort of a top-level transaction to T_0 is an irreversible step.

The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly. The only transactions which actually access data are the leaves of the transaction tree, and thus they are distinguished as "accesses". The partition $\mathcal{O}$ simply identifies those transactions which access the same object.

A serial system of a given system type is the composition of a set of I/O automata. This set contains a transaction for each *internal* (i.e. non-leaf, non-access) node of the transaction tree, a basic object for each element of $\mathcal{O}$ and a serial scheduler. These automata are described below. (If X is a basic object associated with an element $\mathfrak{X}$ of the partition $\mathcal{O}$, and T is an access in $\mathfrak{X}$, we write $T \in$ *accesses(X)* and say that "T is an access to X".)

3.1. Transactions

This paper differs from earlier work such as [Ly,Go,We] in that we model the transactions explicitly, as I/O automata. In modelling transactions, we consider it very important not to constrain them unnecessarily; thus, we do not want to require that they be expressible as programs in any particular high-level programming language. Modelling the transactions as I/O automata allows us to state exactly the properties that are needed, without introducing unnecessary restrictions or complicated semantics.

A non-access *transaction* T is modelled as an I/O automaton, with the following operations.

Input operations:
- CREATE(T)
- COMMIT(T',v), for T' $\in$ children(T) and v $\in$ V
- ABORT(T'), for T' $\in$ children(T)

Output operations:
- REQUEST−CREATE(T'), for T' $\in$ children(T)
- REQUEST−COMMIT(T,v), for v $\in$ V

The CREATE input operation "wakes up" the transaction. The REQUEST−CREATE output operation is a request by T to create a particular child transaction.[2] The 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 ABORT input operation reports to T the unsuccessful completion of one of its children, without returning any other information. We call COMMIT(T',v), for any v, and ABORT(T') *return* operations for

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

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.

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.[3] 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. Thus, transaction automata are required to preserve well-formedness, as defined below.

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 operation, 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) in α'.

- If π is COMMIT(T',v) or ABORT(T') for a child T' of T, then
  (i) REQUEST−CREATE(T') appears in α' and
  (ii) there is no return operation 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'
  (ii) there is no REQUEST−COMMIT(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 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 repeated notification of the fates of its children, does not receive conflicting information

[3]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. This operation could be made explicit, constraining the scheduler to abort the requesting transaction, without substantively changing the model or results.

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 restrictions on allowable transaction behavior. For example, the model allows a transaction to request to commit without discovering the fate of all subtransactions whose creation it has requested. Also, a transaction can request creation of new subtransactions at any time, without regard to its state of knowledge about subtransactions whose creation it has previously requested. Particular programming languages may choose to impose additional restrictions on transaction behavior. (An example is ARGUS, which suspends activity in transactions until subtransactions complete.) However, our results do not require such restrictions.

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. We depart from the traditional notational distinction between creation of subtransactions and invocations on objects, since the common terminology for access and non-access transactions is of great benefit in unifying the statements and proofs of our results. Thus, a *basic object* X is modelled as an automaton, with the following operations.

Input operations:
    CREATE(T), for T in accesses(X)

Output operations:
    REQUEST−COMMIT(T,v), for T in accesses(X)

The CREATE operation is an invocation of an access to the object, while the REQUEST−COMMIT is a return of a value in response to such an invocation.

As with transactions, while specific objects are left largely unspecified, it is convenient to require that schedules of basic objects satisfy certain syntactic conditions. Thus, each basic object is required to preserve well-formedness, defined below.

Let α be a sequence of operations of basic object X. Then an access T to X is said to be *pending* in α provided that there is a CREATE(T), but no REQUEST−COMMIT for T, in α. We define *well-formedness* for sequences of operations of basic objects recursively. 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 operation, 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) in α', and
  (ii) there are no pending accesses in α'.

- If π is REQUEST−COMMIT for T, then
  (i) there is no REQUEST−COMMIT 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, nor does a creation of a new access occur at a basic object before the previous access has completed (i.e. requested to commit); also, a basic object does not respond more than once to any access. and only responds to accesses that have previously been created.

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. 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, however, is a fully specified automaton, particular to each system type. It runs transactions according to a depth-first traversal of the transaction tree. The serial scheduler can choose nondeterministically to abort any transaction 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. The operations of the serial scheduler are as follows.

Input Operations:
  REQUEST−CREATE(T)
  REQUEST−COMMIT(T,v)

Output Operations:
  CREATE(T)
  COMMIT(T,v)
  ABORT(T)

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, COMMIT and ABORT output operations. Each state s of the serial scheduler consists of four sets:

create−requested(s), created(s), commit−requested(s), and returned(s). The set commit−requested(s) 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, (such as returned(s) in the postcondition for REQUEST−CREATE(T)), it is implicit that the set is the same in s' and s (that returned(s') = returned(s), in this example). Note that here, as elsewhere, we have tried to specify the component as nondeterministically as possible, in order to achieve the greatest possible generality for our results.

- REQUEST−CREATE(T)
  Postcondition:
  create−requested(s) = create−requested(s') $\cup$ {T}

- REQUEST−COMMIT(T,v)
  Postcondition:
  commit−requested(s) = commit−requested(s') $\cup$ {(T,v)}

- CREATE(T)
  Precondition:
  T $\in$ create−requested(s') - created(s')
  siblings(T) $\cap$ created(s') $\subseteq$ returned(s')
  Postcondition:
  created(s) = created(s') $\cup$ {T}

- COMMIT(T,v)
  Precondition:
  (T,v) $\in$ commit−requested(s')
  T $\notin$ returned(s')
  children(T) $\cap$ create−requested(s') $\subseteq$ returned(s')
  Postcondition:
  returned(s) = returned(s') $\cup$ {T}

- ABORT(T)
  Precondition:
  T $\in$ create−requested(s') - created(s')
  siblings(T) $\cap$ created(s') $\subseteq$ returned(s')
  Postcondition:
  created(s) = created(s') $\cup$ {T}
  returned(s) = returned(s') $\cup$ {T}

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 precondiition 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 to its parent 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 run sequentially with respect to their siblings.

3.4. Serial Systems and Serial Schedules

In this subsection, we define serial systems precisely and provide some useful terminology for talking about them.

The composition of transactions with basic objects and the serial scheduler for a given system type is called a *serial system*. Define the *serial operations* to be those operations which occur in the serial system: REQUEST−CREATES, REQUEST−COMMITS, CREATES, COMMITS and ABORTS. The schedules of a serial system are called *serial schedules*. The non-access transactions and basic objects are called the system *primitives*. (Recall that each basic object is an automaton corresponding to a set of access transactions. Thus, individual access transactions are not considered to be primitives.)

Recall that the operations of the basic objects have the same syntax as transaction operations. It is convenient to refer to CREATE(T) and REQUEST−COMMIT(T), when T is an access to basic object X, both as operations of transaction T and of object X. To avoid confusion, it is important to remember that there is no transaction automaton associated with any access operation.

For any serial operation π, we define *transaction(π)* to be the transaction at which the operation occurs. (For CREATE(T) operations and REQUEST−COMMIT operations for T, the transaction is T, while for REQUEST−CREATE(T) operations, and COMMIT and ABORT operations for T, the transaction is parent(T).) For a sequence α of serial operations, transaction(α) is the set of transactions of the operations in α.

Two sequences of serial operations, α and α', are said to be *equivalent* provided that they consist of the same operations, and $\alpha|P = \alpha'|P$ for each primitive P. Obviously, this yields an equivalence relation on sequences of serial operations.

We let $\alpha|T$ denote the subsequence of α consisting of operations whose transaction is T, even if T is an access. (This is an extension of the previous definition of $\alpha|T$, as accesses are not component automata of the serial system.)

Let α be a sequence of serial operations. We say that a transaction T is *live* in α provided that a CREATE(T), but no COMMIT(T,v) or ABORT(T), occurs in α. We say that transaction T' is *visible* to T in α provided that for each ancestor T'' of T' which is a proper descendant of lca(T,T'), some COMMIT(T'',v) occurs in α. (In particular, any ancestor of T is visible to T in α.) For sequence α and transaction T, let *visible*(α,*T*) be the subsequence of α consisting of operations whose transactions are visible to T in α. (These include access transactions T'.) We say that transaction T *sees everything* in α provided that visible(α,T) = α.

This is the same definition of visibility as appears, in a different model, in [Ly]. Visibility captures an intuitive notion suggested by the name: the transactions visible to a transaction T in α are those whose effects T is permitted to "see" in α. If transaction T' is visible to transaction T in α, it means that descendants of T may have passed to T information about T', obtained by accessing objects that were previously accessed by descendants of T'.

If α is a sequence of operations, not necessarily all serial, then define serial(α) to be the subsequence of α consisting of the serial operations. We say that T is *live* in α provided that it is live in serial(α). We say that T' is *visible* to T in α if T' is visible to T in serial(α), and define visible(α,T) to be visible(serial(α),T). Also, T *sees everything* in α provided that T sees everything in serial(α). Similarly, define transaction(α) = transaction(serial(α)).

3.5. Correctness Condition

We use serial schedules as the basis of our correctness definitions. Namely, we say that a sequence of operations is *serially correct for a primitive P* provided that its projection on P is identical to the projection on P of some serial schedule. We say that any sequence of operations is *serially correct* if it is serially correct for every non-access transaction. That is, α "looks like" a serial schedule to every non-access transaction.

In the remainder of this paper, we define two systems: concurrent systems and weak concurrent systems. We show that schedules of concurrent systems are serially correct, and that schedules of weak concurrent systems are serially correct for T_0. Thus, we use the serial scheduler as a way of describing desirable behavior, just as serial schedules describe desirable behavior in more classical concurrency control settings (those without nesting). Then serial correctness plays the role in our theory that serializability plays in classical settings.

Note that our correctness conditions are defined at the transaction interface only, and do not constrain the object interface. We believe that this makes the conditions more meaningful to users, and more likely to suffice for a large variety of algorithms, which may use a variety of back-out, locking or version schemes to implement objects. Previous work has focussed on correctness conditions at the object interface [EGLT, etc.]. While we believe that object interface conditions are important, their proper role in the theory is not to serve as the basic correctness condition. Rather, they are useful as intermediate conditions for proving correctness of particular implementations.

The serial correctness condition says that a schedule α must look like a serial schedule to each non-access transaction; this allows for the possibility that α might look like *different* serial schedules to different non-access transactions. This condition may at first seem to be too weak. It may seem that we should require that all transactions see a projection of the *same* serial schedule. But this stronger condition is not satisfied by most of the known concurrency control algorithms. Also, the serial correctness condition is really not as weak as it may seem at first because T_0, the root transaction, is included among the transactions to which α must appear serial. As discussed above, transaction T_0 can be thought of as modelling the environment in which the rest of the transaction system runs. Its REQUEST−CREATE operations correspond to the invocation of top-level transactions, while its COMMIT and ABORT operations correspond to return values and external effects of those transactions. Since α's projection on T_0 must be serial, the environment of the transaction system will see only results that could arise in a serial execution.

3.6. Properties of Serial Systems

In this subsection, we give a pair of lemmas which describe ways in which serial schedules can be "cut and pasted" to yield other serial schedules. These lemmas are used in the proof of the main theorem, in Section 7. The proofs are omitted from this paper, but appear in [LM]. [LM] also contains many additional interesting properties of the behavior of serial systems.

Lemma 2: Let $\alpha\beta_1$COMMIT(T',u) 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) T' sees everything in $\alpha\beta_1$,
(3) T sees everything in $\alpha\beta_2$,
(4) α = visible($\alpha\beta_1$,T'') = visible($\alpha\beta_2$,T'') and
(5) no basic object has operations in both β_1 and β_2.
Then $\alpha\beta_1$COMMIT(T',u)β_2 is a serial schedule.

Lemma 3: 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) T sees everything in $\alpha\beta$, and
(3) α = visible(α,T'') = visible($\alpha\beta$,T'').
Then αABORT(T')β is a serial schedule.

4. Resilient Objects

Having stated our correctness conditions, we are now ready to begin describing implementations and proving that they meet the requirements. This section and the next are devoted to the description of a concurrent system which permits the abort of transactions that have performed steps. An important component of a concurrent system is a new kind of object called a "resilient object," which we introduce in this section. A resilient object is similar to a basic object, but it has the additional capability to undo operations of transactions that it discovers have aborted.

Resilient objects have no capabilities for managing concurrency: rather, they assume that concurrency control is handled externally (by lock manager components of the scheduler). This section defines resilient objects. The complete paper [LM] presents some of their properties, and also describes and proves correct a particular implementation of resilient objects, constructed by keeping multiple versions of corresponding basic objects.

4.1. Definitions

Resilient object R(X) mimics the behavior of basic object X, but has two additional input operations, INFORM−COMMIT−AT(X)OF(T) and INFORM−ABORT−AT(X)OF(T), for every transaction T. Upon receiving an INFORM−ABORT−AT(X)OF(T), R(X) erases any effects of accesses which are descendants of T. This property is made formal as the "Resiliency Condition" below.

R(X) has the following operations, which we call *R(X)-operations.*

Input Operations:
  CREATE(T), T an access to X
  INFORM−COMMIT−AT(X)OF(T)
  INFORM−ABORT−AT(X)OF(T)

Output Operations:
  REQUEST−COMMIT(T,v), T an access to X

In order to describe well-formedness for resilient objects, we require a technical definition for the set of transactions which are *active* after a sequence of R(X)-operations. Roughly speaking, the transactions which are active are those on whose behalf the object has carried out some activity, but whose fate the object does not know.

The definition is recursive on the length of the sequence of R(X) operations. Namely, only T_0 is active after the empty sequence. Let $\alpha = \beta\pi$, where π is a single operation, and let A and B denote the sets of active transactions after α and β, respectively. If π is CREATE(T), then $A = B \cup \{T\}$. If π is a

REQUEST−COMMIT for T, then A = B. If π is INFORM−COMMIT−AT(X)OF(T), and if T is in B, then A = (B - {T}) ∪ {parent(T)}; if T is not in B, then A = B. If π is INFORM−ABORT−AT(X)OF(T), then A = B - descendants(T).

Now we define *well-formedness* for sequences of R(X) operations. Again, the definition is recursive. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of R(X)-operations, 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) in α',
  (ii) all the transactions which are active after α' are ancestors of T.

- If π is a REQUEST−COMMIT for T, then
  (i) there is no REQUEST−COMMIT for T in α', and
  (ii) T is active after α'.

- If π is INFORM−COMMIT−AT(X)OF(T), then
  (i) there is no INFORM−ABORT−AT(X)OF(T) in α', and
  (ii) if T is an access to X, then a REQUEST−COMMIT 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) in α'.

An immediate consequence of these definitions is that the transactions active after any well-formed sequence of R(X)-operations α are a subset of the ancestors of a single active transaction, which we denote least(α).

For α a sequence of R(X)-operations, define *undo(α)* recursively as follows. Define undo(λ) = λ, where λ is the empty sequence. Let $\alpha = \beta\pi$, where π is a single operation. If π is a serial operation (a CREATE or a REQUEST−COMMIT), then undo(α) = undo(β)π. If π is INFORM−COMMIT−AT(X)OF(T), then undo(α) = undo(β). If π is INFORM−ABORT−AT(X)OF(T), then undo(α) is the result of eliminating, from undo(β), all operations whose transactions are descendants of T. Note that undo(α) contains only serial operations.

Let α be any sequence of R(X)-operations, and let π be an operation in α of the form INFORM−ABORT−AT(X)OF(T). Then the *scope* of π in α is the subsequence γ of α consisting of operations eliminated by π.

Resiliency Condition

Resilient object R(X) *satisfies the resiliency condition* if for every well-formed schedule α of R(X), undo(α) is a schedule of basic object X.

We require that resilient object R(X) preserve well-formedness and satisfy the resiliency condition.

The resiliency condition is the correctness condition required by the concurrent schedulers at the object interface. The well-formedness requirement is a syntactic restriction, and the condition that undo(α) be a schedule of basic object X expresses the required semantic relationship between the resilient object and the basic object it incorporates; specifically, that the resilient object "backs out" operations in the scope of INFORM−ABORTS.

5. Concurrent Systems

As with serial schedules in classical settings, our serial schedules contain no concurrency or resiliency and thus are too inefficient to use in practice. Their importance is solely for defining correctness for transaction systems. In this section, we define a new kind of system called a *concurrent system*. The new system consists of the same transactions as in a serial system, a resilient object R(X) for every basic object X of the serial system, and a concurrent scheduler.

Concurrent systems describe computations in which transactions run concurrently and can be aborted after they have performed some work. The concurrent scheduler has the joint responsibility of controlling concurrency and of seeing that the effects of aborted transactions (and their descendants) become undone. Concurrent systems make use of the roll-back capabilities of resilient objects to make sure that ABORT operations in concurrent systems have the same semantics (so far as the transactions can tell) as they do in serial systems.

Concurrent systems are defined in this section. In the next section, the more permissive "weak concurrent systems" are defined. In Section 7, we prove that the schedules of concurrent systems are serially correct, as a corollary of a weaker correctness property for the weak concurrent system.

5.1. Lock Managers

The scheduler we define is called the *concurrent scheduler*. It is composed of several automata: a *lock manager* for every object X, and a single *concurrent controller*. The job of the lock managers is to insure that the associated object receives no CREATES until the lock manager has received abort or commit information for all necessary preceding transactions. This lock manager models an exclusive locking protocol derived from Moss' algorithm [Mo]. The lock manager has the following operations.

Input Operations:
  INTERNAL−CREATE(T), where T is 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 Operations:
CREATE(T), where T is an access to X

The input operations INTERNAL−CREATE, INFORM−COMMIT and INFORM−ABORT will compose with corresponding output operations of the concurrent scheduler which we will construct in this subsection. The output CREATE operation composes with the CREATE input operation of the resilient object R(X). The lock manager receives and manages requests to access object X, using a hierarchical locking scheme. It uses information about the commit and abort of transactions to decide when to release locks.

Each state s of the lock manager consists of the following three sets of transactions: lock−holders(s), create−requested(s), and created(s). Initially, lock−holders = $\{T_0\}$, and the other sets are empty. The operations work as follows.

- INTERNAL−CREATE(T)
  Postcondition:
  create−requested(s) = create−requested(s') ∪ {T}

- INFORM−COMMIT−AT(X)OF(T)
  Postcondition:
  if T ∈ lock−holders(s') then lock−holders(s) = (lock−holders(s') - {T}) ∪ {parent(T)}

- INFORM−ABORT−AT(X)OF(T)
  Postcondition:
  lock−holders(s) = lock−holders(s') - descendants(T)

- CREATE(T)
  Precondition:
  T ∈ create−requested(s') - created(s')
  lock−holders(s') ⊆ ancestors(T)
  Postcondition:
  lock−holders(s) = lock−holders(s') ∪ {T}
  created(s) = created(s') ∪ {T}

Note that resilient object R(X) and the lock manager for X share the INFORM−ABORT and INFORM−COMMIT input operations. These compose with the output from the concurrent controller defined below.

Thus, the lock manager only sends a CREATE(T) operation on to the object in case all the current lock−holders are ancestors of T. When the lock manager learns about the commit of a transaction T for which it holds a lock, it releases the lock to T's parent. When the lock manager learns about the abort of a transaction T for which it holds a lock, it simply releases all locks held by that transaction and its descendants. Our model provides an exceptionally simple and clear way of describing this important algorithm.

5.2. The Concurrent Controller

The concurrent controller is similar to the serial scheduler, but it allows siblings to proceed concurrently. In order to manage this properly, it interacts with "concurrent objects" (lock managers and resilient objects) instead of just basic objects. The operations are as follows.

Input Operations:
REQUEST−CREATE(T)
REQUEST−COMMIT(T,v)

Output Operations:
CREATE(T), T a non-access transaction
INTERNAL−CREATE(T), T an access transaction
COMMIT(T,v)
ABORT(T)
INFORM−COMMIT−AT(X)OF(T)
INFORM−ABORT−AT(X)OF(T)

Each state s of the concurrent controller consists of five sets: create−requested(s), created(s), commit−requested(s), committed(s), and aborted(s). The set commit−requested(s) is a set of (transaction,value) pairs, and the others are sets of transactions. (As before, we will occasionally write $T \in$ commit−requested(s) for $(T,v) \in$ commit−requested(s) for some v.) All sets are initially empty except for create−requested, which is $\{T_0\}$. Define returned(s) = committed(s) $\cup$ aborted(s). The operations are as follows.

- REQUEST−CREATE(T)
  Postcondition:
  create−requested(s) = create−requested(s') $\cup$ {T}

- REQUEST−COMMIT(T,v)
  Postcondition:
  commit−requested(s) = commit−requested(s') $\cup$ {(T,v)}

- CREATE(T), T a non-access transaction
  Precondition:
  T $\in$ create−requested(s') - created(s')
  Postcondition:
  created(s) = created(s') $\cup$ {T}

- INTERNAL−CREATE(T), T an access transaction
  Precondition:
  T $\in$ create−requested(s') - created(s')
  Postcondition:
  created(s) = created(s') $\cup$ {T}

- COMMIT(T,v)
  Precondition:

$(T,v) \in$ commit−requested(s')
$T \notin$ returned(s')
children(T) $\cap$ create−requested(s') $\subseteq$ returned(s')
Postcondition:
committed(s) = committed(s') $\cup$ {T}

- ABORT(T)
  Precondition:
  $T \in$ (create-requested(s') - created(s')) $\cup$ (commit−requested(s') - returned(s'))
  children(T) $\cap$ create−requested(s') $\subseteq$ returned(s')
  Postcondition:
  created(s) = created(s') $\cup$ {T}
  aborted(s) = aborted(s') $\cup$ {T}

- INFORM−COMMIT−AT(X)OF(T):
  Precondition:
  $T \in$ committed(s')

- INFORM−ABORT−AT(X)OF(T):
  Precondition:
  $T \in$ aborted(s')

The concurrent controller is closely related to the serial scheduler. In place of the serial scheduler's CREATE operations, the concurrent controller has two kinds of operations, CREATE operations and INTERNAL−CREATE operations. The former is used for interaction with non-access transactions, while the latter is used for interaction with access transactions. From the concurrent controller's viewpoint, the two operations are the same; however, our naming convention for operations requires us to assign them different names, since the INTERNAL−CREATE operations are intended to be identified with INTERNAL−CREATE operations of the lock managers (which also have CREATE operations, for interaction with the resilient objects). The precondition on the serial scheduler's CREATE operation which insures serial processing of sibling transactions, does not appear in the concurrent controller. Thus, the concurrent controller may run any number of sibling transactions concurrently, provided their parent has requested their creation.

The concurrent controller's COMMIT operation is the same as the serial scheduler's COMMIT operation (except for a minor difference in bookkeeping). The concurrent controller's ABORT operation is different, however; in addition to aborting a transaction in the way that the serial scheduler does, the concurrent controller has the additional capability to abort a transaction that has actually been created and has carried out some steps. In this particular formulation, aborts occur if the transaction was not created (as with the serial scheduler), or if the transaction has previously requested to commit, and its children have returned. Together with the requirements on the COMMIT operation, this condition insures that all transaction completion occurs bottom-up. In the weak concurrent system to be considered in Section 6, a different, "weak", concurrent controller will be used; it differs from the concurrent controller of this section precisely in not requiring ABORT operations to wait for their transactions (and subtransactions) to complete.

The concurrent controller also has two additional operations not present in the serial scheduler. These operations allow the concurrent controller to forward necessary abort and commit information to the lock managers and resilient objects.

5.3. Concurrent Systems

The composition of transactions, resilient objects and the concurrent scheduler (lock managers and concurrent controller) is the *concurrent system*. A schedule of the concurrent system is a *concurrent schedule*, and the operations of a concurrent system are *concurrent operations*.

A main result of this paper is that every concurrent schedule is serially correct. This will be proved as a corollary of another result, in Section 7.

6. Weak Concurrent Systems

In this section, we define "weak concurrent systems", which are exactly the same as concurrent systems, except that they have a more permissive controller, the "weak concurrent controller". The weak concurrent controller reports aborts to a transaction's parent while there is still activity going on in the aborted transaction's subtree. In this paper, weak concurrent systems are used primarily to provide an intermediate step in proving the correctness of concurrent systems: proving a weaker condition for weak concurrent systems allows us to infer the stronger correctness condition for concurrent systems. However, weak concurrent systems are also of interest in themselves. In a distributed implementation of a nested transaction system, performance considerations may make it important for the system to allow a transaction to abort without waiting for activity in the transaction's subtree to subside. In this case, a weak concurrent system might be an appropriate choice, even though the correctness conditions which they satisfy are weaker. Weak concurrent systems also appears to have further technical use, for example in providing simple explanations of the ideas used in "orphan detection" algorithms [HLMW].

6.1. The Weak Concurrent Controller

In this subsection, we define the weak concurrent controller. As we have already said, it is identical to the concurrent controller except that it has a more permissive ABORT operation. For convenience, we describe the controller here in its entirety. It has the same operations as the concurrent controller:

Input Operations:

REQUEST−CREATE(T)
REQUEST−COMMIT(T,v)

Output Operations:
CREATE(T), T a non-access transaction
INTERNAL−CREATE(T), T an access transaction
COMMIT(T,v)
ABORT(T)
INFORM−COMMIT−AT(X)OF(T)
INFORM−ABORT−AT(X)OF(T)

Each state s of the concurrent controller consists of five sets: create−requested(s), created(s), commit−requested(s), committed(s), and aborted(s). The set commit−requested(s) is a set of (transaction,value) pairs, and the others are sets of transactions. (As before, we will occasionally write T ∈ commit−requested(s) for (T,v) ∈ commit−requested(s) for some v.) All are empty initially except for create−requested, which is $\{T_0\}$. Define returned(s) = committed(s) ∪ aborted(s). The operations are as follows.

- REQUEST−CREATE(T)
  Postcondition:
  create−requested(s) = create−requested(s') ∪ {T}

- REQUEST−COMMIT(T,v)
  Postcondition:
  commit−requested(s) = commit−requested(s') ∪ {(T,v)}

- CREATE(T), T a non-access transaction
  Precondition:
  T ∈ create−requested(s') - created(s')
  Postcondition:
  created(s) = created(s') ∪ {T}

- INTERNAL−CREATE(T), T an access transaction
  Precondition:
  T ∈ create−requested(s') - created(s')
  Postcondition:
  created(s) = created(s') ∪ {T}

- COMMIT(T,v)
  Precondition:
  (T,v) ∈ commit−requested(s')
  T ∉ returned(s')
  children(T) ∩ create−requested(s') ⊆ returned(s')
  Postcondition:
  committed(s) = committed(s') ∪ {T}

- ABORT(T)

Precondition:
T ∈ create-requested(s') - returned(s')
Postcondition:
created(s) = created(s') ∪ {T}
aborted(s) = aborted(s') ∪ {T}

- INFORM−COMMIT−AT(X)OF(T):
Precondition:
T ∈ committed(s')

- INFORM−ABORT−AT(X)OF(T):
Precondition:
T ∈ aborted(s')

Thus, the weak concurrent controller is permitted to abort any transaction that has had its creation requested, and which has not yet returned.

6.2. Weak Concurrent Systems

The composition of transactions, resilient objects and the weak concurrent scheduler (lock managers and weak concurrent controller) is the *weak concurrent system*. A schedule of the weak concurrent system is a *weak concurrent schedule.*

Weak concurrent systems exhibit nice behavior to transactions except possibly to those which are descendants of aborted transactions. Thus, we say that a transaction T is an *orphan* in any sequence α of operations provided that an ancestor of T is aborted in α. In many of the properties we prove for weak concurrent systems, we will have to specify that the transactions involved are not orphans. Orphans have been studied in [Go,Wa,HM].

6.3. Properties of Weak Concurrent Systems

We here give some useful basic properties for weak concurrent schedules. As before, complete proofs and additional results appear in [LM].

Lemma 4: Let α be a weak concurrent schedule. Let R(X) be a resilient object, let T and T' be accesses to R(X), and suppose that T' is not an orphan in α. If an operation π of T precedes an operation π' of T' in α, and π is not in the scope of an INFORM−ABORT in α, then T is visible to T' in α.

Proof: By lock manager properties. ∎

The following is a key lemma.

Lemma 5: Let α be a weak concurrent schedule, and let T be live and not an orphan in α.

1. If T' is a transaction, then visible(α,T)|T' is a prefix of α|T' and a schedule of T'.

2. If R(X) is a resilient object, then visible(α,T)|R(X) is a prefix of undo(α|R(X)) and a schedule of basic object X.

Proof: 1. Immediate from the fact that visible(α,T)|T' is either equal to α|T' or is the empty sequence.

2. By Lemma 4 and properties of visibility. ▮

Finally, we show that, in a weak concurrent schedule, concurrently executing transactions access disjoint sets of resilient objects.

Lemma 6: Let α be a weak concurrent schedule, with transactions T and T' live and not orphans in α. Let T'' = lca(T,T'). Let β = visible(α,T) - visible(α,T'') and β' = visible(α,T') - visible(α,T''). Then no resilient object has operations in both β and β'.

Proof: By lock manager properties. ▮

7. Simulation of Serial Systems by Concurrent Systems

In this section, we prove the main results of this paper, that concurrent schedules are serially correct, and that weak concurrent schedules are correct at T_0. Both these results follow from an interesting theorem about weak concurrent schedules, which says that the portion of any weak concurrent schedule which is visible to a live non-orphan transaction is equivalent to (i.e. looks the same at *all* primitives as) a serial schedule.

The proof of this theorem is quite interesting, as it provides considerable insight into the scheduling algorithm. The proof shows not only that a transaction's view of a weak concurrent schedule is equivalent to *some* serial schedule, but by a recursive construction, it actually produces such a schedule. It is interesting and instructive to observe how the views that different transactions have of the system execution get passed up and down the transaction tree, as CREATES, COMMITS and ABORTS occur.

Theorem 7: Let α be a weak concurrent schedule, and T any transaction which is live and not an orphan in α. Then there is a serial schedule β which is equivalent to visible(α,T).

Proof: We proceed by induction on the length of α. The basis, length 0, is trivial. Fix α of length at least 1, and assume that the claim is true for all shorter weak concurrent schedules. Let π be the last operation of α, and let $\alpha = \alpha'\pi$. Fix T which is live and not an orphan in α. We must show that there is a serial schedule β which is equivalent to visible(α,T).

If π is not a serial operation, then visible(α',T) = visible(serial(α'),T) = visible(serial(α),T) = visible(α,T), and the result is immediate by induction. So we can assume that π is a serial operation. Also, if transaction(π) is not visible to T in α, then visible(α,T) = visible(α',T), and the result is again immediate by induction. Thus, we can assume that transaction(π) is visible to T in α. Also, T is not an orphan in α'.

There are four cases.

(1) π is an output operation of a transaction or resilient object.

Then the inductive hypothesis implies the existence of a serial schedule β' which is equivalent to visible(α',T). Let $\beta = \beta'\pi$. We must show that β is equivalent to visible(α,T) and serial.

Let P be any primitive. Then $\beta|P = \beta'\pi|P$ = visible(α',T)π|P by inductive hypothesis, = visible(α,T)|P. Therefore, β is equivalent to visible(α,T).

Let π be an output of primitive P. Then $\beta|P$ = visible(α,T)|P by equivalence, which is a schedule of P by Lemma 5. Lemma 1 implies that β is serial.

(2) π is a CREATE(T') operation.

Then transaction(π) = T', and so T' is visible to T in α. Then π is the first operation in α whose transaction is a descendant of T'. By the definition of visibility, it must be that T' = T. Then parent(T) is live in α'. Since parent(T) is not an orphan in α, the inductive hypothesis implies the existence of a serial schedule β' which is equivalent to visible(α',parent(T)). Let $\beta = \beta'\pi$. We must show that β is equivalent to visible(α,T) and serial.

Let P be any primitive. Then $\beta|P = \beta'\pi|P$, = visible(α',parent(T))π|P by inductive hypothesis, = visible(α,T)|P. Thus, β is equivalent to visible(α,T).

Consider any execution of the serial system having β' as its operation sequence, and let s' be the state of the serial scheduler after β'. Also consider any execution of the weak concurrent system having α as its operation sequence, and let s be the state of the weak concurrent scheduler after α'. Since π is enabled in s, it is easy to show that it is also enabled in s'.

(3) π is a COMMIT(T',v) operation.

Then T'' = parent(T') = transaction(π) is visible to T and not an orphan in α. Also, T' is not an orphan in α'. Then T' is live in α', and so T'' is live in α' and so in α. Since T'' is live and visible to T, T'' is an ancestor of T. Since T is live in α, T is not a descendant of T'. The inductive hypothesis yields two serial schedules, β' and β'', which are equivalent to visible(α',T') and visible(α',T), respectively. Let γ = visible(β',T''). Let $\beta_1 = \beta' - \gamma$ and $\beta_2 = \beta'' - \gamma$. We must show that $\beta = \gamma\beta_1\pi\beta_2$ is equivalent to visible(α,T) and serial.

Equivalence is straightforward. We show that β is serial. This follows from Lemma 2, provided we can show that:
(3.a) $\gamma\beta_1\pi$ is a serial schedule,
(3.b) T' sees everything in $\gamma\beta_1$,
(3.c) T sees everything in $\gamma\beta_2$,
(3.d) γ = visible($\gamma\beta_1$,T'') = visible($\gamma\beta_2$,T'') and
(3.e) no basic object has operations in both β_1 and β_2.
But (3.a) - (3.d) are straightforward, while (3.e) is immediate from Lemma 6.

(4) π is an ABORT(T') operation.

Then T'' = parent(T') = transaction(π) is visible to T in α, and so is not an orphan in α. Then T' is live in α', and T'' is live in α' and so in α. Since T'' is live and visible to T in α, T is a descendant of T''. Since T is not an orphan in α, T is not a descendant of T'. The inductive hypothesis yields two serial schedules, β' and β'', which are equivalent to visible(α',T'') and visible(α',T), respectively. Let $\beta_1 = \beta'' - \beta'$. We must show that $\beta = \beta'\pi\beta_1$ is equivalent to visible(α,T) and serial.

Equivalence is straightforward. We show that β is serial. This follows from Lemma 3, provided we can show that:
(4.a) $\beta'\pi$ is a serial schedule,
(4.b) T sees everything in $\beta'\beta$, and
(4.c) β' = visible(β',T'') = visible($\beta'\beta$,T'').
But this is straightforward. ▮

Corollary 8: Every weak concurrent schedule is serially correct for every non-orphan non-access transaction.

Corollary 9: Every weak concurrent schedule is serially correct for T_0.

Corollary 10: Every concurrent schedule is serially correct.

8. Acknowledgments

We thank Bill Weihl for many, many comments and questions, and much encouragement, during the course of this project. We also thank all the other members of the ARGUS design and implementation group at MIT, for providing a concrete model for us to try to abstract and generalize. Also, we thank Yehuda Afek for his comments on an early draft, and Sharon Perl for her comments on later drafts.

9. References

[AM] Allchin, J. E., and McKendry, M. S., "Synchronization and Recovery of Actions," *Proc. 1983 Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August 17-19, 1982, pp. 31-44.

[BBG] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," Manuscript.

[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.

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

[Go] Goree, Jr., John A., "Internal Consistency of a Distributed Transaction System With Orphan Detection," MS Thesis, Technical Report MIT/LCS/TR-286, MIT Laboratory for Computer Science, Cambridge, MA., 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.

[HM] Herlihy, M., and McKendry, M., "Time-Driven Orphan Elimination", in *Proc. of the 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA., January 1986, pp. 42-48.

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

[KS] Kedem, Z., and Silberschatz, A., "A Characterization of Database Graphs Admitting a Simple Locking Protocol", *Acta Informatica* 16 (1981) pp. 1-13.

[LaS] Lampson, B. W., and Sturgis, H. E., "Crash Recovery in a Distributed Data Storage System," Tech. Rep., Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, Calif., 1979.

[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* 5, 3, (July 1983), pp. 381-404.

[LM] Lynch, N., and Merritt, M., "Introduction to the Theory of Nested Transactions", MIT Technical Report, AT&T Bell Labs Technical Report.

[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.

[Mi] Milner, R., "A Calculus of Communicating Systems", *Lecture Notes in Computer Science*, #92, Springer-Verlag, Berlin, 1980.

[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.

[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.

[RLS] Rosenkrantz, D. J., Lewis, P. M., and Stearns, R. E., "System Level Concurrency Control for Distributed Database Systems," *ACM Transactions on Database Systems*, Vol. 3, No. 2, June 1978, pp. 178-198.

[Wa] Walker, E. F., "Orphan Detection in the Argus System," M.S. Thesis, Technical Report/MIT/LCS/TR-326, MIT Laboratory for Computer Science, Cambridge, MA., June 1984.

[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.

This work 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 Grant DCR-83-02391, and by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-83-K-0125.