CONCURRENCY CONTROL FOR RESILIENT NESTED TRANSACTIONS

Nancy A. Lynch

ABSTRACT

Concurrency control theory is extended to handle nested transactions with failures. The theory is used to present a rigorous correctness proof of a variant of Moss' locking algorithm for implementing nested transactions. The proof has an interesting structure using many levels of abstraction.

1. INTRODUCTION

In the past few years, there has been considerable research on concurrency control, including both systems design and theoretical study. The problem is roughly as follows. Data in a large (centralized or distributed)

Advances in Computing Research, Volume 3, pages 335–373

ISBN: 0-89232-611-5

database is assumed to be accessible to users via *transactions*, each of which is a sequential program which can carry out many steps accessing individual data objects. It is important that the transactions appear to execute "atomically" (i.e., without intervening steps of other transactions). However, it is also desirable to permit as much concurrent operation of different transactions as possible, for efficiency. Thus it is not generally feasible to insist that transactions run completely serially. A notion of *equivalence* for executions is defined, where two executions are equivalent provided they "look the same" to all transactions and to all data objects. The *serializable* executions are just those which are equivalent to serial executions. One goal of concurrency control design is to ensure that all executions of transactions be serializable.

Several characterization theorems have been proved for serializability; generally, they amount to the absence of cycles in some relation describing the dependencies among the steps of the transactions. A very large number of concurrency control algorithms have been devised. Typical algorithms are those based on two-phase locking [3], and those based on timestamps [6]. Although many of these algorithms are very different from each other, they can all be shown to be correct concurrency control algorithms. The correctness proofs depend on the absence-of-cycles characterizations for serializability.

More recently, it has been suggested [7, 8, 10] that some additional structure on transactions might be useful for programming distributed databases, and even for programming more general distributed systems. The suggested structure permits transactions to be nested. Thus a transaction is not necessarily a sequential program, but rather can consist of (sequential or concurrent) subtransactions. The intention is that the subtransactions are to be serialized with respect to each other, but the order of serialization need not be completely specified by the writer of the transaction. This flexibility allows more concurrency in the implementation than would be possible with a single-level transaction structure consisting of sequential transactions. The general structure allows transactions to be nested to any depth, with only the leaves of the nesting tree actually performing accesses to data.

Transactions are often used not only as a unit of concurrency, but also as a unit of recovery. In a nested transaction structure, it is natural to try to localize the effects of failures within the closest possible level of nesting in the transaction nesting tree. One is naturally led to a style of programming which permits a transaction to create children, and to tolerate the reported failure of some of its children, using the information about the occurrence of the failures to decide on its further activity. The intention is that failed transactions are to have no effect on the data or on other transactions. This style of programming is a generalization of the

"recovery block" style of [9] to the domain of concurrent programming. Indeed, this style seems to be especially suitable for programming distributed systems, since many types of failures of pieces of programs are likely to occur in such systems.

Reed [10] has designed an algorithm which uses multiple versions of data to implement nested transactions. Moss [8] has abstracted away from Reed's specific implementation of nested transactions, presenting a general description of the nested transaction model. He has also developed an alternative implementation of the nested transaction model, based on two-phase locking. This model and implementation are fundamental to the Argus distributed computing language, now under development by Liskov's group at MIT [7].

The basic correctness criteria for nested transactions seem to be clear enough, intuitively, to allow implementors a sufficient understanding of the requirements for their implementation. However, some subtle issues of correctness have arisen in connection with the behavior of failed subtransactions. For example, the Argus group has decided that a pleasant property for an implementation to have is that all transactions, including even "orphans" (subtransactions of failed transactions), should see "consistent" views of the data (i.e., views that could occur during an execution in which they are not orphans). The implementation goes to considerable lengths to try to ensure this property, but it is difficult for the implementors to be sure that they have succeeded.

It seems clear that some basic groundwork is needed before such properties can be proved. Namely, the theory already developed for concurrency control of single-level transaction systems without failures needs to be generalized to incorporate considerations of nesting and failures. The model needs to be formal, in order to allow careful specification of all the correctness requirements—the simple and intuitive ones, as well as the rather subtle ones.

This paper begins to develop this groundwork. First, a simple "action tree" structure is defined, which describes the ancestor relationships among executing transactions and also describes the views which different transactions have of the data. A generalization of serializability to the domain of nested transactions with failures is defined. A characterization is given for this generalization of serializability, in terms of absence of cycles in an appropriate dependency relation on transactions. A slightly simplified version of Moss' algorithm is presented in detail, and a correctness proof is given.

The correctness proof is complete, detailed, and rigorous. Its style appears to be quite interesting in its own right. Producing such a proof was a very difficult task; the main issues that made it so difficult were the nesting of transactions and the possible failures of subtransactions. The

initial attempts to develop such a proof led to extremely complicated, nonmodular constructions. Gradually, after we had tried for many months to organize the proof, the uniform general proof structure presented in this paper began to emerge. This structure allows the proof to be decomposed in a very natural way. Without this structure, it is doubtful that we would have been able to complete a proof at all. (We know of few comparably successful complete proofs for difficult distributed algorithms.)

The proof is based on certain algebras, which we call "event-state" algebras. An event-state algebra is an abstract description of a computing system and the protocol that governs its behavior. The elements of the algebra are states of the computing system. An operation of the algebra is an "event" of the system (i.e., a computation step); it transforms a state to another state. The operations are only partially defined, in correspondence with the fact that a step might not be applicable to all states. The rules that specify when an operation is defined correspond to the algorithm or protocol that controls the execution of the system.

Another important concept for our proof is the notion of a mapping between algebras. It is useful to describe a computing system on several different levels of abstraction (i.e., as several distinct algebras). A mapping from an algebra $\mathscr{A}$ to another algebra $\mathscr{B}$ is a "simulation" of $\mathscr{B}$ by $\mathscr{A}$ provided that every valid computation of $\mathscr{A}$ is mapped to a valid computation of $\mathscr{B}$. Thus $\mathscr{A}$ is, in a sense, an "implementation" of $\mathscr{B}$.

The approach taken in this paper to a correctness proof of Moss' algorithm is the following. The system governed by the algorithm is described by a succession of algebras, each one describing more specific details about the algorithm and its implementation. In the highest level algebra, the only precondition for the applicability of a step (an operation) is that it preserve global correctness. This algebra is quite far from the algorithm itself. As a matter of fact, this algebra represents "what needs to be achieved" by the system. Successive algebras get closer to the algorithm, that is, to "how it is achieved". Showing the existence of a simulation mapping between each pair of successive levels is the heart of the correctness proof.

One novel aspect of the simulations we use, different from the usual notions of "abstraction" mappings, is that our simulations map single lower level states to *sets* of higher level states, rather than just single higher level states. (We call them "possibilities" mappings.) This extra flexibility seems quite convenient for many implementations, allowing the lower level algebra sometimes to contain less detail than the higher level algebra. For example, it might be easy to prove correctness of an algorithm which maintains lots of auxiliary data. The correctness of an algorithm which contains less detail could be proved, in our model, by

showing that it simulates the algorithm which maintains the auxiliary data.

While possibilities mappings are convenient for proving correctness of ordinary centralized algorithms, they produce their greatest payoff for distributed algorithms. Namely, a distributed algorithm is described as a special case of an event-state algebra, a "distributed algebra." A distributed algebra has a set of "components." The state set for the algebra is just a Cartesian product of local states, one for each component. The events are partitioned among the set of components, according to which a component is assumed to "perform" the event. Event domains and transitions are defined componentwise. To show that a distributed algebra simulates some other "abstract" algebra, it suffices to define an appropriate possibilities mapping from the global states of the distributed algebra to sets of states of the abstract algebra. It turns out to be extremely natural to describe such a mapping by first describing a possibilities mapping from the local state of each component to sets of abstract states. The image of a local state under this mapping just represents the set of possible global states consistent with the knowledge of the particular component. The possibilities for the entire distributed algebra are simply obtained by taking the intersection of the possibilities consistent with the knowledge of all the components.

It appears that this technique extends to give natural proofs of many algorithms, especially distributed algorithms, and thus warrants further investigation. Goree [4] presents a slightly more general development of the technique than is presented in this paper, but more remains to be done.

The concurrency control definitions given in this paper express the most fundamental correctness requirements, but not subtle conditions such as correctness of orphans' views. Issues of fairness and eventual progress are not addressed, but rather only safety properties, serializability in particular. Future work involves extending the framework presented here to allow expression of these other properties and to allow correctness proofs for the difficult algorithms which guarantee these properties. Some further work in these directions has already been carried out: Goree [4] gives a definition for correctness of orphans' views and has given a correctness proof for a complicated algorithm used in the implementation of Argus to maintain correctness of orphans' views in the face of transaction aborts.

A related recent paper [1] also addresses the problem of proving correctness of algorithms implementing nested transactions. However, that paper does not address issues of failure and recovery, which are primary considerations of the present paper. Also, the kind of nesting they consider appears to be somewhat different from ours: it appears to

be designed primarily for describing levels of data abstraction. Finally, the proof techniques of [1] are quite different from ours.

Although our variant of Moss' algorithm is described completely in this paper, we urge the interested reader to read Moss' presentation in [8]. His presentation gives useful background and context for the algorithm, as well as a much more intuitive description of the algorithm than is presented here.

2. EVENT-STATE ALGEBRAS

In this section, we describe the event-state algebra framework. This framework is used in the later sections to organize the formal correctness proof for Moss' algorithm. The algorithm is described in a series of five levels, each of which is described as an event-state algebra.

The reader who is mainly interested in the formal model for nested transactions, and in Moss' algorithm, rather than in proofs of concurrent algorithms, can safely skim the contents of this section.

2.1. Algebras and Simulations

We begin with the basic algebra definitions. An *event-state algebra* $\mathscr{A} = \langle A, \sigma, \Pi \rangle$, consists of a set A of *states*, an element $\sigma \in A$, the *initial state*, and a set Π of partial unary operations (the *events*). In this paper, we will usually refer to an event-state algebra as simply an *algebra*.

Next, we give standard definitions for computability concepts. For any event π, we let *domain*(π) denote the set of states for which π is defined. Let a be a state, and let $\Phi = (\pi_1, \ldots, \pi_k)$ be any finite sequence of events chosen from Π. Then Φ is said to be *valid* from a provided $b = \pi_k(\pi_{k-1}(\ldots(\pi_1(a))\ldots))$ is defined (i.e., provided that $\pi_{i-1}(\ldots(\pi_1(a))\ldots)$ is in *domain*(π_i), for all i, $1 \leq i \leq k$). In this case, b is called the *result* of Φ applied to a. An infinite sequence of events is said to be *valid* from a provided all its finite prefixes are valid from a. We say that Φ is *valid* provided it is valid from σ, and the *result* of Φ is defined to be the result of Φ applied to σ. We write $a \vdash b$ provided there is some finite Φ, valid from a, for which b is the result of Φ applied to a. b is *computable* provided $\sigma \vdash b$.

In order to decompose our proof into levels of abstraction, we require a definition of "simulation" of an algebra $\mathscr{A} = \langle A, \sigma, \Pi \rangle$ by another algebra $\mathscr{A}' = \langle A', \sigma', \Pi' \rangle$. In this paper, we present a very weak definition. An *interpretation* of $\mathscr{A}$ by $\mathscr{A}'$ is a mapping $h: \Pi' \to \Pi \cup \{\Lambda\}$. (Here, Λ represents a null event.) We extend h to a homomorphism mapping event sequences of $\mathscr{A}'$ to event sequences of $\mathscr{A}$ in the obvious way

(deleting occurrences of Λ). An interpretation h is a *simulation* of $\mathscr{A}$ by $\mathscr{A}'$ provided that $h(\Phi')$ is a valid event sequence for $\mathscr{A}$ whenever Φ' is a valid event sequence for $\mathscr{A}'$.

We note that these definitions do not rule out certain trivial situations. We have not imposed the general requirement that $\mathscr{A}'$ include a representation of every event in $\mathscr{A}$. We have also not imposed any requirements that events of $\mathscr{A}'$ be defined on large domains. Thus our techniques are not powerful enough to prove that $\mathscr{A}'$ does everything which is required to implement $\mathscr{A}$ correctly; rather, we assume that $\mathscr{A}'$ is given, and we are to prove that everything it does is correct for $\mathscr{A}$. We believe that the more powerful techniques required to ensure the stronger properties require extra machinery and a more sophisticated general theory than we wish to present here.

The first lemma gives a basic composition result. This lemma justifies our composition of simulation results for adjacent levels to prove a simulation result for nonadjacent levels.

LEMMA 1. *Assume that $\mathscr{A}$, $\mathscr{A}'$, and $\mathscr{A}''$ are algebras, that h is a simulation of $\mathscr{A}$ by $\mathscr{A}'$ and h' is a simulation of $\mathscr{A}'$ by $\mathscr{A}''$. Then $h \circ h'$ is a simulation of $\mathscr{A}$ by $\mathscr{A}''$.*

PROOF. Straightforward. □

2.2. Possibilities Mappings

Our basic method for proving correctness is showing that simulations exist between adjacent members of a sequence of algebras. Therefore, we need a tool that can be used to show that a mapping is a simulation. In this subsection, we give a sufficient condition for a mapping h from $\mathscr{A}'$ to $\mathscr{A}$ to be a simulation. The condition involves defining a correspondence between states of the two algebras, in addition to events. It turns out to be most convenient, for the reasons discussed in the Introduction, to allow the state mapping to map a single state of $\mathscr{A}'$ to a set of states of $\mathscr{A}$ rather than just to a single state. The states in such a set are called "possibilities" (i.e., the "possible" states corresponding to a given state). If we think of $\mathscr{A}'$ as a "concrete" algebra and $\mathscr{A}$ as a more "abstract" algebra, then we see that a possibilities mapping allows single "concrete" states to be mapped to sets of "abstract" states rather than to just single abstract states.

Let $h: A' \cup \Pi' \to \mathscr{P}(A) \cup \Pi \cup \{\Lambda\}$ be such that $h(a') \in \mathscr{P}(A)$ for all $a' \in A'$, and h restricted to Π' is an interpretation, that is, $h(\pi') \in \Pi \cup \{\lambda\}$ for all $\pi' \in \Pi'$. (Here, $\mathscr{P}$ denotes the power set.) Then h is a *possibilities*

mapping from $\mathcal{A}'$ to $\mathcal{A}$ provided the following are true:

(a) $\sigma \in h(\sigma')$.

Assume a and a' are computable in $\mathcal{A}$ and $\mathcal{A}'$, respectively, and $a \in h(a')$. Assume $\pi' \in \Pi'$. Assume $a' \in \mathit{domain}(\pi')$ and $b' = \pi'(a')$.

(b) If $h(\pi') = \pi \in \Pi$, then $a \in \mathit{domain}(\pi)$.
(c) If $h(\pi') = \pi \in \Pi$, then $\pi(a) \in h(b')$.
(d) If $h(\pi') = \Lambda$, then $a \in h(b')$.

Property (a) says that the initial state of $\mathcal{A}$ is among the possibilities for the initial state of $\mathcal{A}'$. Property (b) says that an event is only performed in $\mathcal{A}'$ when its image event can be performed in $\mathcal{A}$. Properties (c) and (d) say that events performed in $\mathcal{A}'$ preserve possibilities. Figure 1 should be helpful in understanding (b) and (c). A similar diagram can be drawn to illustrate (d).

The following lemmas show that any possibilities mapping is a simulation.

LEMMA 2. *Let h be a possibilities mapping from $\mathcal{A}'$ to $\mathcal{A}$. If Φ' is a valid event sequence for $\mathcal{A}'$ and $h(\Phi') = \Phi$, then Φ is a valid event sequence for $\mathcal{A}$. In addition, if Φ' is finite, a' is the result of Φ' and a is the result of Φ, then $a \in h(a')$.*

PROOF. By induction on the length of Φ'. □

LEMMA 3. *Any possibilities mapping from $\mathcal{A}'$ to $\mathcal{A}$ is a simulation of $\mathcal{A}$ by $\mathcal{A}'$.*

PROOF. Immediate by Lemma 2. □

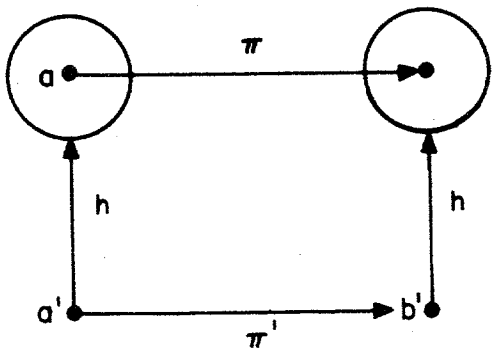

Figure 1. A property of possibilities maps.

2.3. Distributed Algebras

Next, we define a special kind of event-state algebra, called a "distributed algebra." A distributed algebra is one which can be decomposed into components in a simple way: the states are Cartesian products of states for the components, each event is assumed to be originated by some particular component (although it can affect other components), and the definability and effects of events are locally determined. Such an algebra provides a natural structure for describing distributed algorithms. Processors in a network and message systems are typical examples of components in such a decomposition.

An algebra $\mathscr{A} = \langle A, \sigma, \Pi \rangle$ is said to be *distributed* over a finite index set I using d, provided that A is the Cartesian product of sets A_i, $i \in I$, d is a mapping, d: $\Pi \rightarrow I$, giving the "doer" of each event, and the following two conditions are satisfied.

- (Local Domain) Let $i = d(\pi)$. If a, $b \in A$ and $a_i = b_i$, then $a \in$ *domain*(π) if and only if $b \in$ *domain*(π).
- (Local Changes) If a, $b \in$ *domain*(π), $a' = \pi(a)$, $b' = \pi(b)$, and $a_i = b_i$, then $a'_i = b'_i$.

The local domain property says that the state of the doer of an event determines the definability of that event. The local change property says that the changes caused by an event are defined componentwise. Note that in the local change property, the component i need not necessarily be the doer of π; we permit other components to be affected by π, but assume that the effect is uniquely determined by π and the state of the component. Strictly speaking, we could have omitted mention of both of these properties in this paper, since they are not needed to prove the one simple result we obtain (Lemma 4) about distributed algebras. However, the properties seem to describe the locality structure of distributed algorithms quite accurately, and so we present them in anticipation of further study.

It happens that there is a particularly natural way to define a possibilities mapping from a distributed algebra to another algebra. Namely, we define a "local mapping" from the local state of each component of the distributed algebra to a set of abstract states. The result of this mapping should be thought of as the set of possible abstract states, as far as a particular component can tell from its local knowledge. The mapping from a global state of the distributed algebra can then be defined to yield the intersection of the images of all the component states. The conditions we require for local mappings are chosen to be sufficient to guarantee that the derived global mapping is a possibilities mapping.

Let $\mathcal{A}' = \langle A', \sigma', \Pi' \rangle$ be an algebra, distributed over I using d. Let $\mathcal{A} = \langle A, \sigma, \Pi \rangle$ be an algebra. Let h be an interpretation from $\mathcal{A}'$ to $\mathcal{A}$. For each $i \in I$, let h_i: $A' \to \mathcal{P}(A)$ be such that h_i depends on A'_i only; that is if $a_i = b_i$, then $h_i(a) = h_i(b)$. Then we say that h and h_i, $i \in I$, form a *lcoal mapping* from $\mathcal{A}'$ to $\mathcal{A}$ provided the following conditions are satisfied.

(a) For all $i \in I$, $\sigma \in h_i(\sigma')$.

Fix any $i \in I$ (for properties (b)–(d)). Assume a and a' are computable in $\mathcal{A}$ and $\mathcal{A}'$, respectively, and $a \in h_i(a')$. Assume $\pi' \in \Pi$, $d(\pi') = i$. Assume $a' \in \mathit{domain}(\pi')$ and $b' = \pi'(a')$.

(b) If $h(\pi') = \pi \in \Pi$, then $a \in \mathit{domain}(\pi)$.

Fix (for properties (c) and (d)) any $j \in I$. (This j can be the same as or different from i.)

(c) Assume $h(\pi') = \pi \in \Pi$ and $a \in h_j(a')$. Then $\pi(a) \in h_j(b')$.
(d) Assume $h(\pi') = \Lambda$ and $a \in h_j(a')$. Then $a \in h_j(b')$.

That is, (a) says that the initial state of $\mathcal{A}$ is in the set of possibilities for each component's initial state. Property (b) says that an event is only performed in $\mathcal{A}'$ when its doer knows that its image event can be performed in $\mathcal{A}$. Properties (c) and (d) consider the situation from the point of view of an arbitrary component j. Property (c) says that an event with doer i preserves possibilities at component j. Property (d) is analogous to (c) for events whose images are null events.

Figure 2 illustrates property (b), and Figure 3 illustrates property (c).

The following lemma shows that local mappings yield possibilities mappings.

LEMMA 4. *Let $\mathcal{A}$ and $\mathcal{A}' = \langle A', \sigma', \Pi' \rangle$ be algebras, where $\mathcal{A}'$ is*

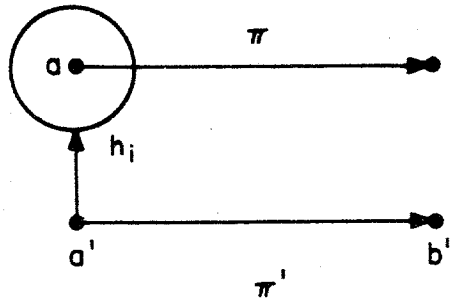

Figure 2. A property of local mappings.

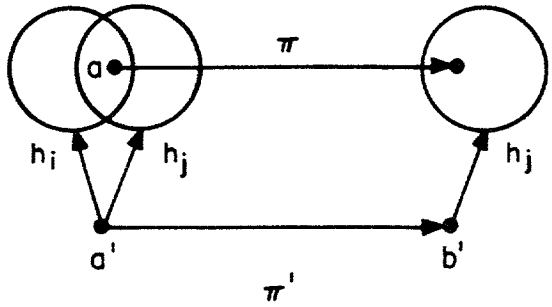

Figure 3. Another property of local mappings.

distributed over I. Assume that h and h_i, $i \in I$, form a local mapping from $\mathscr{A}'$ to $\mathscr{A}$. Extend h to $A' \cap \Pi'$ by defining $h(a') = \bigcap_{i \in I} h_i(a')$. Then h is a possibilities mapping from $\mathscr{A}'$ to $\mathscr{A}$ (and therefore a simulation of $\mathscr{A}$ by $\mathscr{A}'$).

PROOF. We check the four properties of the possibilities mapping definition.

(a) To see that $\sigma \in h(\sigma')$, it suffices to show that $\sigma \in h_i(\sigma')$ for all $i \in I$. But this is exactly the statement of property (a) of the local mapping definition.

Now we assume the hypotheses supplied for parts (b)–(d) of the possibilities mapping definition. Assume also that $d(\pi') = i$.

(b) Since $a \in h(a')$, it is obvious that $a \in h_i(a')$. Property (b) of the local mapping definition implies that $a \in domain(\pi)$.

(c) In order to show that $\pi(a) \in h(b')$, it suffices to fix an arbitrary $j \in I$ and show that $\pi(a) \in h_j(b')$. Since $a \in h_j(a')$, the needed property follows from (c) of the local mapping definition.

(d) It suffices to show that $a \in h_j(b')$ for any $j \in I$. This follows as in the preceding argument from (d) of the local mapping definition. □

If the definitions in this section are to be used in correctness proofs for the widest possible class of algorithms, they will probably need to be generalized. In particular, it seems appropriate to permit single events of a more concrete algebra to interpret sequences of events of a more abstract algebra. (See Goree [4] for definitions and uses for this generalization.) Also, allowing each algebra to have a set of initial states rather than just a single initial state would probably be useful. Since we do not need these generalizations here, we do not make these extensions.

3. ACTION TREES

In this section, we provide the basic definitions needed to describe properties of nested transactions. The definitions in this section describe a particular data structure, called an "action tree," which provides a natural representation of nested transactions, the relationships between them, and their views of data. We define "serializability" in terms of action trees. We also prove several very basic lemmas about the definitions.

We caution the reader that there are many definitions in this section, and he should not try to remember them all. Rather, we suggest that he read the definitions once for familiarity, and then use the section for later reference.

In the rest of the paper, we often refer to transactions as just "actions," for brevity. This departure from the usual conventions of database theory has been made for consistency with the Argus work.

3.1. Objects and Actions

The system is assumed to contain a set of data objects, upon which the nested actions operate. We begin with some definitions for objects. Let *obj* be a universal set of data objects. For each $x \in obj$, let $values(x)$ denote the set of values x can assume, including a distinguished initial value $init(x)$. A *value assignment* is a total mapping f from *obj* to $values(obj)$, having the property that $f(x) \in values(x)$ for all $x \in obj$.

Next, we give basic definitions for actions. In this paper, we have chosen to avoid modeling transactions explicitly with a particular programming model. Rather, we have attempted to extract from such a model just that information which is needed for concurrency control theory.

Let *act* be a universal set of actions. Let U be a distinguished action. We assume that the actions are configured a priori into a tree representing their nesting relationship, with U as the root. For every $A \in act - \{U\}$, let $parent(A)$ denote a unique parent action for A. Let *siblings* denote $\{(A, B) \in act^2 : parent(A) = parent(B)\}$. If $A \in act$, let $children(A)$ denote $\{B \in act : parent(B) = A\}$. If $A, B \in act$, let $lca(A, B)$ denote the least common ancestor of A and B. If $A \in act$, let $desc(A)$ (resp. $anc(A)$) be the set of descendants (resp. ancestors) of A. Let $proper\text{-}desc(A)$ (resp. $proper\text{-}anc(A)$) be the set of proper descendants (resp. ancestors) of A.

It might be convenient for the reader to think of this as an a priori configuration of all possible actions into a tree as a preassigned "naming scheme" for actions. That is, the "name" of any action is assumed to

carry within it information which locates that action in this universal tree of actions. In any particular execution, only some of these possible actions will be "activated." The (virtual) action U, the parent of all top-level actions, has been added for the sake of uniformity. Its presence provides a simplification in many arguments.

We assume a priori determination of which actions actually access data, which objects they access and the functions they perform on those objects. Namely, let *accesses* denote the leaves of the tree described above. It is exactly these actions which access data. (We assume that $U \notin accesses$, so that the entire set of actions is nontrivial.) Let $object: accesses \rightarrow obj$ be a fixed function. If $object(A) = x$, we say that A *is an access to* x. For $A \in accesses$, let $update(A): values(object(A)) \rightarrow values(object(A))$ be a fixed function, describing the change made by A to its object. Let *sameobject* denote $\{(A, B) \in accesses^2: object(A) = object(B)\}$.

It might at first appear that our model does not permit updates to depend on previous steps executed by a transaction. This is not our intention. Dependence on previous steps is modeled by our choice of a particular access: the "name" of the access is assumed to carry information about previous steps executed by a transaction.

Note that the usual read and write operations of serializability theory can be regarded as special cases of accesses. Namely, "read accesses" have the identity function as their associated update function, while "write accesses" have an associated update function which is a constant function.

3.2. Action Trees

Next, we give a way of describing a "snapshot" of a particular execution, using a structure called an "action tree." An action tree can be regarded as the generalization of the log from ordinary serializability theory. The information captured in an action tree includes which actions have been "activated," what the status of each such action is (i.e., active, committed, or aborted), and what value of its data object was seen by each access.

An *action tree* T has components $vertices_T$, $active_T$, $committed_T$, $aborted_T$, and $label_T$, where

- $vertices_T$ is a finite subset of *act*, closed under the parent operation: if $A \in vertices_T - \{U\}$, then $parent(A) \in vertices_T$. (These represent the actions which have ever been created during the current execution.)
- $active_T$, $committed_T$, and $aborted_T$ comprise a partition of $vertices_T$.

(These classifications indicate the current status of each action that has ever been created. When a nonaccess action is first created, it is classified as active. At some later time, its classification can be changed to either committed or aborted. By "committed" we mean that the action is committed relative to its parent, but not necessarily committed permanently. Permanent commit of an action would be represented by classification of all ancestors of the action, except for U, as committed. Section 3.4 contains definitions and a lemma about permanent commit of actions.)

- $label_T$: $datasteps_T \to values(obj)$ (where $datasteps_T = committed_T \cap accesses$), with $label_T(A) \in values(object(A))$. (The label of an access to an object is intended to represent the value read by that access. Since the access has an associated function, the value which the access writes into the object is deducible from the value read, and therefore need not be explicitly represented. As a technical convenience, we do not assign a label to accesses until they become committed.)

The following definitions are just convenient shorthand for concepts already defined. Let $done_T$ denote $committed_T \cup aborted_T$. Let $status_T$ be defined by $status_T(A) =$ 'active' (resp. 'committed', 'aborted') provided $A \in active_T$ (resp. $committed_T$, $aborted_T$). Let $accesses_T = vertices_T \cap accesses$, $accesses_T(x) = \{B \in accesses_T : object(B) = x\}$, and $datasteps_T(x) = \{B \in datasteps_T : object(B) = x\}$.

3.3. Visibility

Next, we give a very important definition which helps to describe the "views" which actions have of each other and of the data. In particular, this definition allows us to describe actions whose existence is intended to be known to other actions (i.e., not masked from those other actions by intervening failures or active actions). For $A \in vertices_T$, let $visible_T(A)$ denote $\{B \in vertices_T : anc(B) \cap proper\text{-}desc(lca(A, B)) \subseteq committed_T\}$. That is, $visible_T(A)$ is just the set of actions whose existence is potentially known to action A, because they and all their ancestors, up to and not including some ancestor of A, have committed (to their parents). Action A will be permitted to see the results of updates made by the transactions in $visible_T(A)$, and no others. For $A \in vertices_T$, $x \in obj$, let $visible_T(A, x)$ denote $visible_T(A) \cap datasteps_T(x)$. The following lemma describes elementary properties of "visibility."

LEMMA 5. *Let T be an action tree, $A, B, C \in vertices_T$.*

a. *If* $B \in desc(A)$, *then* $A \in visible_T(B)$.
b. $A \in visible_T(B)$ *if and only if* $A \in visible_T(lca(A, B))$.
c. *If* $A \in visible_T(B)$ *and* $B \in visible_T(C)$, *then* $A \in visible_T(C)$.
d. *If* $A \in desc(B)$ *and* $C \in visible_T(B)$, *then* $C \in visible_T(A)$.
e. *If* $A \in desc(B)$ *and* $A \in visible_T(C)$, *then* $B \in visible_T(C)$.

PROOF

a. Immediate
b. Immediate from the fact that $lca(A, B) = lca(A, lca(A, B))$.
c. Let $D \in anc(A) \cap proper\text{-}desc(lca(A, C))$. We must show that $D \in committed_T$. If $D \in proper\text{-}desc(lca(A, B))$, then the fact that $A \in visible_T(B)$ implies the result. So assume that $D \notin proper\text{-}desc(lca(A, B))$. It must be the case that $D \in anc(lca(A, B))$, and that $lca(B, C) = lca(A, C)$. Thus $D \in anc(B) \cap proper\text{-}desc(lca(B, C))$, so the fact that $B \in visible_T(C)$ implies the result.
d. Immediate from parts a and c.
e. Immediate from parts a and c. □

A related definition allows us to describe actions which are capable of "committing up to the top level." If $A \in vertices_T$, then we say A *is live in* T provided $anc(A) \cap aborted_T = \emptyset$, and we say A *is dead in* T otherwise.

LEMMA 6. *If* $A, B \in vertices_T$, A *is live in* T, *and* $B \in visible_T(A)$, *then* B *is live in* T.

PROOF. If B is dead in T, then there exists $C \in anc(B) \cap aborted_T$. We know $C \notin proper\text{-}desc(lca(A, B))$, since $B \in visible_T(A)$. Thus $C \in anc(lca(A, B)) \subseteq anc(A)$, so A is dead in T, a contradiction. □

3.4. Serializability

In this subsection, we develop the basic correctness condition for action trees: serializability.

First, we define the result of applying a sequence of steps to a data object. If $x \in obj$ and s is a finite sequence of datasteps, then we define $result(x, s)$ as follows. If s is the empty sequence, then $result(x, s) = init(x)$. Otherwise, let $s = s'A$. Then $result(x, s) = update(A)(result(x, s'))$ if A involves x, and $result(x, s) = result(x, s')$ otherwise.

If S is a set, and $\leq$ is a total order on the elements of S, then we let

$\langle\langle S; \leq \rangle\rangle$ denote the sequence consisting of the elements of S, in the order given by $\leq$.

In order to define serializability, we need to consider linear orderings of all sets of siblings in the action tree. Thus let T be an action tree. A partial order $p \subseteq$ siblings is *linearizing* for T provided p totally orders all sets of siblings in T. A linearizing partial order p induces a total order $induced_{T,p}$, on $datasteps_T$, in the obvious way; if A and B are datasteps, with respective ancestors A' and B', where A' and B' are siblings, then $(A, B) \in induced_{T,p}$ if and only if $(A', B') \in p$. If $A \in datasteps_T(x)$ and p is a linearizing partial order for T, let $preds_{T,p}(A)$ denote $\langle\langle \{B \in visible_T(A, x): (B, A) \in induced_{T,p} \text{ and } B \neq A\}; induced_{T,p} \rangle\rangle$. Thus, $preds_{T,p}(A)$ denotes the sequence of datasteps whose effects on A's object are supposed to be visible to A.

A linearizing partial order p for T is said to be a *serializing* partial order for T provided that $label_T(A) = result(x, preds_{T,p}(A))$ for all $A \in datasteps_T(x)$. That is, the value actually seen by A for its data object is exactly the result of the datasteps whose effects are supposed to be visible to A. T is said to be *serializable* provided there exists some serializing partial order for T.

In this paper, we consider serializability of portions of an action tree rather than an entire action tree. In particular, it might sometimes be useful to require serializability only for those actions whose effects become "permanent," and not worry about those which get aborted.

Thus, given an action tree T, a new action tree, $perm(T)$, is defined as follows.

- $vertices_{perm(T)} = visible_T(U)$. (Lemma 5e shows that $perm(T)$ is a tree.)
- $A \in vertices_{perm(T)}$, then $status_{perm(T)}(A) = status_T(A)$. (This status is always "committed," except for U.)
- If $A \in datasteps_{perm(T)}$, then $label_{perm(T)}(A) = label_T(A)$.

The following lemma shows the useful property that all the vertices in a permanent subtree are visible to each other.

LEMMA 7. *If T is an action tree and $A, B \in vertices_{perm(T)}$, then $B \in visible_{perm(T)}(A)$.*

PROOF. Since $B \in vertices_{perm(T)} = visible_T(U)$, Lemma 5d implies that $B \in visible_T(A)$. Then $B \in visible_{perm(T)}(A)$, since the status of each vertex is the same in T and $perm(T)$. □

In this paper, we will use the correctness condition that any tree T created by our algorithm should have $perm(T)$ serializable. It is worth noting that one of the reasons that actions might be aborted is that a concurrency controller has discovered that allowing an action to proceed or commit will corrupt serializability. Thus there is no reason to expect complete action trees to be serializable, and we focus on the permanent part of the trees only.)

3.5. Discussion

Note that the style in which serializability is defined here constrains the implementation less than the type of definition used in "traditional" concurrency control theory. The earlier definitions regard the data as external to the concurrency control algorithm; the algorithm is to take requests for data accesses and translate them into actual accesses, observing appropriate rules. Generally, the accesses performed by the concurrency control algorithm simply obtain the latest version of the data object. A clue that the earlier definitions are too constraining is that they do not apply unchanged to algorithms, such as Reed's, which use sophisticated management of versions of the data. The earlier definitions require extensions [2, 5] to handle algorithms such as Reed's. These extensions still regard the data as external to the concurrency control algorithm, and so the modified correctness conditions contain explicit information about particular "versions" of the data objects. It seems, however, that the *appearance* of serializability, in terms of the values seen by the accesses, is really all that matters—it is possible that this appearance could be preserved by some algorithm which does not operate in terms of versions at all.

The less constraining approach which is taken here is to regard the data as internal to the concurrency control algorithm, at least for the purpose of stating the basic correctness conditions. Thus the definitions introduced in this paper are intended to be applicable to algorithms which use single versions of data objects, algorithms that use multiple versions of data objects, as well as to other implementation as yet unforeseen.

4. AN ALGEBRA BASED ON ACTION TREES

In this section, we begin to use the event-state algebra framework. We use the set of action trees as the state set for an algebra and define a set of standard events which we would like to allow to be performed on action trees. We describe each event by defining the circumstances under

which the event is to be allowed to be performed (the "precondition") and the resulting changes to be made in the action tree (the "effect").

We will use this algebra as a specification of correct abstract system behavior, the first level in our correctness proof. Thus we must ensure that the definition of this algebra includes the property that all action trees it generates have their permanent subtrees serializable. One way of doing this would be to include preservation of serializability explicitly in all the preconditions. It is a little simpler notationally just to state the serializability condition as a global invariant, to be maintained by all events; thus we follow this latter option. In terms of the algebraic model, there is an implicit precondition on each event stating that the result of the event satisfies the global invariant.

We now define a set of events on action trees. That is, we define an algebra $\mathscr{A} = \langle A, \sigma, \Pi \rangle$, where A is the set of action trees, σ is the trivial action tree with the single vertex U, with status 'active', and Π contains the four kinds of events described in (a)–(d) below. We define the events as follows. First, we let C denote the set of all action trees T for which $perm(T)$ is serializable. (In particular, $\sigma \in C$.) We place an implicit precondition on each event, stating that the result of the event is in C. Within this constraint, we define the domain by giving a precondition on action trees T and use assignment notation to describe the effect of the event on T.

In all events, we assume that $A \in act - \{U\}$.

(a) **create**$_A$
- (a1) Precondition
  - (a11) $A \notin vertices_T$.
  - (a12) $parent(A) \in vertices_T - committed_T$.
- (a2) Effect
  - (a21) $vertices_T \leftarrow vertices_T \cup \{A\}$.
  - (a22) $status_T(A) \leftarrow$ 'active'.

(b) **commit**$_A$, $A \notin$ accesses
- (b1) Precondition
  - (b11) $A \in active_T$.
  - (b12) $children(A) \cap vertices_T \subseteq done_T$.
- (b2) Effect
  - (b21) $status_T(A) \leftarrow$ 'committed'.

(c) **abort**$_A$
- (c1) Precondition
  - (c11) $A \in active_T$.
- (c2) Effect
  - (c21) $status_T(A) \leftarrow$ 'aborted'.

(d) **perform**$_{A,u}$, $A \in$ **accesses**, $x = \text{object}(A)$, $u \in values(x)$
  (d1) Precondition
    (d11) $A \in active_T$.
  (d2) Effect
    (d21) $status_T(A) \leftarrow$ 'committed'.
    (d22) $label_T(A) \leftarrow u$.

The meaning of the four events is as follows. The **create**$_A$ event creates or ("activates") a new action. It is required, of course, that A not be already in the tree. Its parents must be there, however, and must not already be committed (since a committed parent is assumed to have all of its children completed and to depend on the completion of the particular set of children it had at the time of commit). Note that we allow A to be created after its parent has aborted. This might be reasonable in an implementation in which the two events occur at different nodes of a distributed system, for example. The effect of creating A is to add A to the tree, with status 'active'.

The **commit**$_A$ event commits an active nonaccess action. It requires that A be active and all its children be completed. The effect is to change the status to 'committed'. The **abort**$_A$ event is similar, but there is no requirement on the children—an active action can abort at any time.

Finally, the **perform**$_{A,u}$ event actually performs a step on a data object. It requires that *access* A be active and changes its status to 'committed'. It also records (in our action tree analog to the "log") the value u seen by the access. (It is unnecessary to record the value written, since that could be inferred from the value seen.) Note that we do not specify how the value u is supposed to be obtained by the **perform** event; it is permissible to record any value, as long as the serializability condition is preserved.

We note that the only events which could cause the serializability constraint to be violated are commit and perform events. Thus these are the only events for which the implicit precondition C is actually necessary.

We also note that this algebra provides considerable flexibility in allowable sequences of events.

5. AUGMENTED ACTION TREES

Now, we proceed to the second level of our proof. As before, it will be useful to define a data structure first and then develop an algebra based on that data structure. The data structure to be used in the second level

is called an "augmented action tree." It is very similar to an action tree, but includes some extra information describing a sequence of versions for each data object. An augmented action tree is similar to a transaction conflict graph with resolution of conflicts. We stated earlier that we did not want to rely on definitions that depend on data versions for our basic correctness conditions. However, the definitions which make specific reference to versions are still useful in conjunction with the approach of this paper. Their role is in supplying sufficient conditions for serializability, and thereby helping to organize correctness proofs.

Serializability is defined for agumented action trees. It is seen that serializability for augmented action trees implies serializability for corresponding action trees. Moreover, serializability for augmented action trees has a cycle-free characterization similar to those in usual concurrency control theory. Therefore, this structure can be useful in proofs of serializability for action trees.

Thus it is at our second level that the interesting concurrency control arguments occur.

5.1. Augmented Action Tree Definitions

An *augmented action tree* (AAT) T is a pair $(S, data_T)$, where S is an action tree and $data_T \subseteq sameobjects_S$ is a partial order on $datasteps_S$ which totally orders the datasteps for each object. We extend action tree notation to T; for example, we write $datasteps_T$ to denote $datasteps_S$. We also extend the definitions of *visible*, *live*, *dead*, *linearizing*, *induced*, *preds*, and *serializable* to T by applying them to S.

The assumed ordering on accesses to each data object imposes an ordering on siblings higher up in the tree. If T is an AAT, then let $sibling\text{-}data_T$ denote $\{(A, B) \in siblings: (C, D) \in data_T$ for some $C \in desc(A)$, $D \in desc(B)\}$.

We require notation for an access' visible predecessors in the version order. If $A \in datasteps_T(x)$, then let $v\text{-}data_T(A)$ denote $\{B \in visible_T(A, x)$: $(B, A) \in data_T$ and $B \neq A\}$. The following is a technical lemma.

LEMMA 8. *Let T be an AAT. Let p be a linearizing partial order for T, $x \in obj$, and $A \in datasteps_T(x)$. Assume that $induced_{T,p}$ is consistent with $data_T$. Then $preds_{T,p}(A) = \langle\langle v\text{-}data_T(A); data_T \rangle\rangle$.*

PROOF. Straightforward. □

An AAT T is *data-serializable* provided there exists p, a serializing partial order for T, with the additional property that $induced_{T,p}$ is

consistent with $data_T$. Thus T is data-serializable provided that it is serializable in a way that respects the conflict resolution partial ordering. Of course, data-serializability for AAT's provides a sufficient condition for serializability.

5.2. Characterization of Data-Serializability

The analog of the usual characterization in concurrency control theory is proved in this subsection. Namely, we give a characterization of data-serializability in terms of absence of cycles.

First, we give a definition which says that the label of each access describes the correct object value which the access should see if the versions of objects are ordered according to the $data_T$ order. Formally, an AAT is *version-compatible* provided for every $x \in obj$ and every $A \in datasteps_T(x)$, it is the case that $label_T(A) = result(x, s)$, where $s = \langle\langle v\text{-}data_T(A); data_T \rangle\rangle$.

The next theorem contains the characterization result.

THEOREM 9. *An AAT T is data-serializable if and only if both of the following are true:*

a. *T is version-compatible.*
b. *There are no cycles of length greater than one in $sibling\text{-}data_T$.*

PROOF. Assume T is data-serializable and obtain p, a serializing partial order for T for which $induced_{T,p}$ is consistent with $data_T$.

a. Let $A \in datasteps_T(x)$, $s = \langle\langle v\text{-}data_T(A); data_T \rangle\rangle$. Then $label_T(A) = result(x, preds_{T,p}(A))$, by the definition of serializability, $= result(x, s)$, by Lemma 8.
b. $sibling\text{-}data_T \subseteq p$. Thus there are no cycles of length greater than one in $sibling\text{-}data_T$.

Now assume a and b. Let p be any partial order which totally orders all siblings and is consistent with $sibling\text{-}data_T$. Then p is linearizing for T, and $induced_{T,p}$ is consistent with $data_T$. We will show that p is a serializing partial order for T. Let $x \in obj$, $A \in datasteps_T(x)$. We must show that $label_T(A) = result(x, preds_{T,p}(A))$. Since T is version-compatible, we know that $label_T(A) = result(x, s)$, *where* $s = \langle\langle v\text{-}data_T; data_T \rangle\rangle$. Then Lemma 8 implies that $s = preds_{T,p}(A)$, as needed. □

6. AN ALGEBRA BASED ON AUGMENTED ACTION TREES

In this section, we define the algebra for our second level. This algebra will be based on the set of AAT's. We define events on AAT's analogously to the definitions for action trees. Once again, we carry out the definitions within the event-state algebra framework. We then prove several basic properties of this algebra. Finally, we show that this algebra simulates the level 1 algebra.

The second-level algebra can be understood as describing the "abstract effect" achieved by locking algorithms. (We do not actually describe a locking mechanism until later levels.) The major accomplishment of this section involves showing that this abstract effect in fact guarantees the required serializability condition. The argument is relatively nontrivial and is analogous to the usual correctness proofs for strict two-phase locking. Argument for later levels will show that locking protocols actually achieve the required abstract effect. Thus we have factored the correctness proof for a locking algorithm into two natural parts.

6.1. Definitions

We define a new algebra $\mathscr{A}' = \langle A', \sigma', \Pi' \rangle$, where A' is the set of AAT's, σ' is the trivial AAT which has a single vertex U with status 'active', and the events of Π' correspond closely to the events of $\mathscr{A}$, and are designated by the same names. (We will rely on context to distinguish the two cases.) The only differences are that there is no global constraint corresponding to C, and $\textbf{perform}_{A,u}$ introduces two additional preconditions and an additional change. These new conditions can be thought of as capturing the abstract effect of a variant of Moss' locking algorithm.

(d1) Precondition

(d12) Let $B \in datasteps_T(x)$, B live in T. Then $B \in visible_T(A, x)$.

(d13) If A is live in T, then $u = result(x, s)$, where $s = \langle\langle visible_T(A, x);\ data_T \rangle\rangle$.

(d2) Effect

(d23) $data_T \leftarrow data_T \cup \{(B, A): B \in datasteps_T(x)\} \cup \{(A, A)\}$.

The new preconditions say that a data access A must wait long enough so that all live accesses to the object have been committed, up to the level which matters to A. Also, the value used in the access is just the

one resulting from the sequence of previous accesses in the given data ordering. The new effect just involves adding appropriate new pairs to the end of the data ordering.

6.3. Preliminary Results

This section contains two straightforward lemmas. The first describes some invariants preserved by the events.

LEMMA 10. *If T is computable in $\mathscr{A}'$, then the following are true.*

a. *If $A \in vertices_T$ and $parent(A) \in committed_T$, then $A \in done_T$.*
b. *$U \in active_T$.*
c. *If $(B, A) \in data_T$, then either B is dead in T, or else $B \in visible_T(A)$.*
d. *If $A \in committed_T$ and $B \in desc(A) \cap vertices_T$, then either B is dead in T else $B \in visible_T(A)$.*

PROOF. Most of the arguments are straightforward. We argue cases c and d.

c. If $B = A$, the result is immediate. If $B \neq A$, then the only way we get $(B, A) \in data_T$ is by virtue of some $\mathbf{perform}_{A,u}$ event. That is, there exists T' such that $T' \vdash T$, such that the precondition for some step $\mathbf{perform}_{A,u}$ is satisfied in T'. Thus B is dead in T' or $B \in visible_T(A)$. Therefore, B is dead in T or $B \in visible_T(A)$.
d. If $B = A$, the result is immediate. So assume $A \neq B$. Let $A \in committed_T$, $B \in desc(A) \cap vertices_T$, B live in T, and $B \notin visible_T(A)$. Then there exist C, $D \in desc(A) \cap anc(B)$, for which $C = parent(D)$, $C \in committed_T$ and $D \in active_T$. But this contradicts part a. □

The second lemma of this subsection describes properties that hold of a pair of AAT's, one of which is derivable from the other.

LEMMA 11. *Let T and T' be computable in $\mathscr{A}'$, and assume that $T \vdash T'$.*

a. *$vertices_T \subseteq_{T'}$, $committed_T \subseteq committed_{T'}$, $aborted_T \subseteq aborted_{T'}$, and $data_T \subseteq data_{T'}$.*
b. *If $A \in datasteps$, then $label_T(A) = label_{T'}(A)$.*
c. *If $A \in datasteps_T$ and $(B, A) \in data_{T'}$, then $(B, A) \in data_T$.*
d. *If $A \in vertices_T$, then $visible_T(A) \subseteq visible_{T'}(A)$.*

e. *If* $A \in vertices_T$ *and* A *is live in* T', *then* A *is live in* T.
f. *If* $A = parent(B)$ *and* $A \in committed_T$ *and* $B \in vertices_{T'}$, *then* $B \in done_T$.

PROOF. The only case that takes some arguing is f. Let $A = parent(B)$, $A \in committed_T$ and $B \in vertices_{T'}$. Let T' be the result of ϕ applied to T, and let T be the result of Ψ. Then Ψ contains a step π of the form $\mathbf{commit}_A$, and $\Psi\Phi$ contains a step ρ of the form $\mathbf{create}_B$. π cannot precede ρ, since the precondition for ρ would be violated. So ρ precedes π. Then the precondition for π implies that $B \in done_T$. □

6.3. Computability Guarantees Data-Serializability

Note that there is no correctness condition for AAT's explicitly mentioning serializability. This is because for AAT's, computability alone is sufficient to guarantee serializability of $perm(T)$, as we show in the next theorem. It is convenient to prove the two required properties separately in two lemmas. The second of these two lemmas is the hardest result in the paper.

LEMMA 12. *If* T *is computable in* $\mathcal{A}'$, *then* $perm(T)$ *is version-compatible*.

PROOF. Let $A \in datasteps_{perm(T)}(x)$. We must show that u $(= label_{perm(T)}(A)) = result(x, s)$, where $s = \langle\langle v\text{-}data_{perm(T)}(B); data_{perm(T)}\rangle\rangle$. A is inserted into the tree by a $\mathbf{perform}_{A,u}$ step π, so let the event sequence producing T be written as $\Phi\pi\Psi$. Let T' denote the result of Φ, and T'' the result of $\Phi\pi$. The preconditions for π show that $label_{T''}(A) = result(x, s')$, where $s' = \langle\langle visible_{T'}(A, x); data_{T'}\rangle\rangle$. By Lemma 11b and the definition of $perm(T)$, it follows that $label_{perm(T)}(A) = result(x, s')$. Thus it suffices to show that $s = s'$. Since both $data_{T'}$ and $data_{perm(T)}$ are consistent with $data_T$, it suffices to show that s and s' contain the same elements.

First, let $B \in s$. Then $(B, A) \in data_T$ and so by Lemma 11c, $B \in datasteps_{T''}(x)$. Since A is the only element in T'' which is not in T', $B \in datasteps_{T'}(x)$. Since $A \in vertices_{perm(T)} = visible_T(U)$, and $U \notin aborted_T$ (by Lemma 10), it follows that A is live in T. Since $B \in visible_T(A)$, Lemma 6 shows that B is live in T. Thus B is live in T', by Lemma 11e. The precondition for π implies that $B \in visible_{T'}(A, x)$, so $B \in s'$.

Conversely, suppose $B \in s'$. Then $B \neq A$ since $A \notin vertices_{T'}$. Then $(B, A) \in data_{T''}$, so by Lemma 11a, $(B, A) \in data_T$. By Lemma 11d, $B \in visible_T(A, x)$. By Lemma 7, it suffices to show that $B \in vertices_{perm(T)} = visible_T(U)$. But $B \in visible_T(A)$ and $A \in visible_T(U)$, so Lemma 5c suffices. □

LEMMA 13. *If T is computable in $\mathcal{A}'$, then there are no nontrivial cycles in sibling-data$_{perm(T)}$.*

PROOF. Assume the contrary: let $(\sigma = A_0, A_1, \ldots, A_k = \sigma)$, $k \geq 2$, be a minimum length cycle such that $(A_i, A_{i+1}) \in sibling\text{-}data_{perm(T)}$ for all i, $0 \leq i \leq k-1$. Let a sequence Φ of events be defined so that T is the result of Φ. We will show that for each i, $0 \leq i \leq k-1$, there exists a prefix Ψ_i of Φ such that if T' is the result of Ψ_i, then $A_i \in done_{T'}$, and $A_{i+1} \notin done_{T'}$. If we fix i for which Ψ_i is of maximum length and let T' be the result of this Ψ_i, then we see that $A_{i+1} \notin done_{T'}$. But Ψ_{i+1} is no longer than Ψ_i, so Lemma 11a implies that $A_{i+1} \in done_{T'}$, which is a contradiction.

So fix i, $0 \leq i \leq k-1$. Then $(A_i, A_{i+1}) \in sibling\text{-}data_{perm(T)}$. Then there exist $B \in desc(A_i)$, $C \in desc(A_{i+1})$ with $(B, C) \in data_{perm(T)}$. Since $B, C \in vertices_{perm(T)}$, it follows that $(anc(B) \cup anc(C)) \cap proper\text{-}desc(U) \subseteq committed_T$. Now, Φ has a prefix $\Psi\pi$, where π is a **perform**$_{C,u}$ step, Let T' be the result of Ψ, and T'' the result of $\Psi\pi$. Lemma 11c implies that $(B, C) \in data_{T''}$, so that $B \in datasteps_{T'}$. Since B is live in T (using Lemma 10b), Lemma 11e implies that B is live in T'. Then the precondition for π implies that $B \in visible_{T'}(C)$, which means that $A_i \in anc(B) \cap proper\text{-}desc(lca(B, C)) \subseteq committed_{T'} \subseteq done_{T'}$. We must show that $A_{i+1} \notin done_{T'}$; if we can do this, then taking $\Psi_i = \Psi$ yields the result. Assume $A_{i+1} \in done_{T'}$. Then let D be the lowest ancestor of C for which $D \in done_{T'}$: it must be the case that $D \in anc(C) \cap proper\text{-}desc(lca(B, C)) \subseteq committed_T$, so $D \in committed_{T'}$. Since $C \in active_{T'}$, we know that $D \neq C$. Let E be the single element of $children(D) \cap anc(C)$. Then $E \notin done_{T'}$. Then $E \notin vertices_T$ by Lemma 11f. This means $C \notin vertices_T$. This is contradiction. □

THEOREM 14. *If T is computable in $\mathcal{A}'$, then $perm(T)$ is data-serializable.*

PROOF. Immediate from Lemma 12, Lemma 13, and Theorem 9. □

6.4. Simulation

Next, we show that $\mathcal{A}'$ simulates $\mathcal{A}$. We define a mapping h from $\mathcal{A}'$ to $\mathcal{A}$ as follows. If $T = (S, data_T)$ is an AAT, then $h(T) = \{S\}$. If π is in Π', then $h(\pi)$ is just the event in Π with the same name.

LEMMA 15. *h is a simulation of $\mathcal{A}$ by $\mathcal{A}'$.*

PROOF. (a) and (d) of the definition of a possibilities mapping are immediate. Property (b) follows immediately from the fact that $a' \in$

domain(π') (since only additional constraints are added for $\mathscr{A}'$); note that Theorem 14 implies that the C-constraint is always satisfied. Property (c) is then straightforward. Thus, h is a possibilities mapping. Lemma 3 shows that h is a simulation. □

7. AN ALGEBRA BASED ON VERSION MAPS

In order to complete the proof of Moss' algorithm, it remains to prove that it achieves the abstract effect of locking described by $\mathscr{A}'$. It seems simplest to decompose this task further, first showing that a centralized locking algorithm simulates $\mathscr{A}'$, and then showing that a distributed version of the algorithm simulates the centralized version. It turns out to be feasible to decompose the proof of the centralized locking algorithm still further. Namely, we first describe a locking-style algorithm which retains a large amount of useful information. Then we show that a more optimized locking algorithm simulates the algorithm which retains information.

In this section, we develop the third level of the algorithm, the locking-style algorithm which retains information.

7.1. Version Maps

As before, we begin by introducing another data structure, called a "version map." This one records some locking information for each object. As in Moss' algorithm, each object has a stack of locks, held at any time by a sequence of actions which are successive descendants. The version map records, for each object, and each action in some sequence of successive descendants, the sequence of accesses to the object whose result is available to the action.

Thus a *version map* is a partial mapping V from *obj x act* to sequences of accesses, such that the following properties are satisfied:

- $V(x, U)$ is defined for all x.
- Each $V(x, A)$ consists of accesses to x.
- For each x, if $V(x, A)$ and $V(x, B)$ are both defined, then either $A \in desc(B)$ or $B \in desc(A)$.
- If $V(x, A)$ and $V(x, B)$ are both defined and $B \in desc(A)$, then $V(x, B)$ is an extension of $V(x, A)$.

Thus, for each x, V is defined only for transactions which lie on some chain of ancestors; V is not necessarily defined for all transactions on the chain, but only for some subset of the transactions on the chain.

If A is the least action for which $V(x, A)$ is defined, then we call A the *principal action* for x in V; in this case, if $result(x, V(x, A)) = u$, we say that u is the *principal value* of x in V.

7.2. Definition of the Algebra

We define another algebra $\mathscr{A}'' = \langle A'', \sigma'', \Pi'' \rangle$ as follows. A'' is the set of pairs (T, V), where T is an AAT and V is a version map. σ'' consists of the trivial AAT consisting of a single node U with status 'active', and the version map which has $V(x, U)$ equal to the empty sequence, for all x, and is otherwise undefined. Π'' consists of the six events defined below in (a)–(f).

In all the events to follow, we assume that $A \in act - \{U\}$. Events (a)–(c) are identical to (a)–(c) of $\mathscr{A}'$. Some changes are needed in the **perform** event, and there are two new events which manipulate locks.

(d) **perform**$_{A,u}$, $A \in accesses$, $x = object(A)$, $u \in values(x)$
- (d1) Precondition
  - (d11) $A \in active_T$.
  - (d12) $\{B: V(x, B) \text{ is defined}\} \subseteq proper\text{-}anc(A)$.
  - (d13) u is the principal value of x in V.
- (d2) Effect
  - (d21) $status_T(A) \leftarrow$ 'committed'.
  - (d22) $label_T(A) \leftarrow u$.
  - (d23) $data_T \leftarrow data_T \cup \{(B, A): B \in accesses_T(x)\} \cup \{(A, A)\}$.
  - (d24) $V(x, A) \leftarrow V(x, B) \circ (A)$, where B is the principal action in V.

(e) **release-lock**$_{A,x}$, $x \in obj$
- (e1) Precondition
  - (e11) $V(x, A)$ is defined.
  - (e12) $A \in committed_T$.
- (e2) Effect
  - (e21) $V(x, parent(A)) \leftarrow V(x, A)$.
  - (e22) $V(x, A) \leftarrow$ undefined.

(f) **lose-lock**$_{A,x}$, $x \in obj$
- (f1) Precondition
  - (f11) $V(x, A)$ is defined.
  - (f12) A is dead in T.
- (f2) Effect
  - (f21) $V(x, A) \leftarrow$ undefined.

Thus (d) says that a **perform**$_{A,u}$ event can only be carried out when the

current lock-holders are all proper ancestors of A and when u is the proper value which should be provided to A. This event has the new effect of augmenting the version map by giving a "lock" to A: A gets a sequence of versions which is exactly that held by the previous principal action, concatenated with a new version for A. Event (e) allows a lock to be released by a committed action; its effect is to pass the lock up to its parent, so that its parent now obtains the sequence of versions previously held by the child. Event (f) allows a lock to be released by a dead action.

7.3. Basic Properties

In this subsection, we present a simple lemma stating some important invariants preserved in $\mathscr{A}''$.

LEMMA 16. *If (T, V) is computable in $\mathscr{A}''$, then the following are true.*

a. *If $V(x, A)$ is defined, then $A \in vertices_T$.*
b. *If $B \in datasteps_T(x)$ and B is live in T, then there exists $A \in anc(B)$ with $V(x, A)$ defined and B an element of $V(x, A)$.*
c. *If $V(x, A)$ is defined, then each element of $V(x, A)$ is in $visible_T(A)$.*
d. *If $V(x, A)$ is defined, then the elements of $V(x, A)$ are in $data_T$ order.*

PROOF. Straightforward. We argue b, for example. Immediately after an event $\mathbf{perform}_{B,u}$ occurs, we see that $V(x, B)$ is defined, and $B \in V(x, B)$. Assume inductively that there is some ancestor C of B with $V(x, C)$ defined and $B \in V(x, C)$. Since B remains live, there are no steps of the form $\mathbf{lose\text{-}lock}_{C,x}$. Thus if $V(x, C)$ is ever changed, it must be because of a release-lock step. There are two possibilities. First, the change could occur because of a $\mathbf{release\text{-}lock}_{C,x}$ step. But such a step causes $V(x, parent(C))$ to take on the old value of $V(x, C)$, thereby preserving the needed property. Second, the change could occur because $V(x, C)$ gets redefined to be the previous value of $V(x, D)$, where $D \in children(C)$. But because the successive sequences are extensions of each other, B is an element of $V(x, D)$ as well. Thus the needed property is preserved in this case also. □

7.4. Simulation

Define a mapping h' from $\mathscr{A}''$ to $\mathscr{A}'$ as follows, h' maps (T, V) to $\{T\}$, and maps events (a)–(d) to events of the same name, and events (e) and (f) to Λ.

LEMMA 17. *h' is a simulation of $\mathscr{A}'$ by $\mathscr{A}''$.*

PROOF. It suffices to show that h' is a possibilities mapping. Properties (a) and (d) are easy to check. We consider property (b). Let $\pi' \in \Pi''$, where $h'(\pi') = \pi \in \Pi'$. Then π' is either of the form $\mathbf{create}_A$, $\mathbf{commit}_A$, $\mathbf{abort}_A$, or $\mathbf{perform}_{A,u}$. In the first three cases, the property (b) is easy to check. So assume that π' is of the form $\mathbf{perform}_{A,u}$. Assume (T, V) is computable in $\mathscr{A}''$ and π' is defined on (T, V), yielding (T', V'). We must show that $\mathbf{perform}_{A,u}$ (i.e., the event of $\mathscr{A}'$) is defined on T. Let $x = object(A)$.

Condition (d11) for $\mathscr{A}'$ follows immediately from the corresponding condition for $\mathscr{A}''$. We consider (d12). Let $B \in datasteps_T(x)$, and assume that B is live in T. Since (T, V) is computable in $\mathscr{A}''$, Lemma 16 implies that there is some $C \in anc(B)$ for which $V(x, C)$ is defined and for which B is an element of $V(x, C)$. Then Lemma 16 implies that $B \in visible_T(C)$. Since π' is defined on (T, V), (d12) for $\mathscr{A}''$ implies that $C \in anc(A)$. Since $A \in vertices_T$, Lemma 5 implies that $B \in visible_T(A)$, as needed.

Next, we consider (d13). Assume A is live in T, and let $s = \langle\langle visible_T(A, x); data_T \rangle\rangle$. We must show that $u = result(x, s)$. Let B be the principal action for x in V. Condition (d13) for $\mathscr{A}''$ implies that $u = result(x, V(x, B))$. It suffices to show that s and $V(x, B)$ are identical. Since the elements of $V(x, B)$ are in $data_T$ order (by Lemma 16), it suffices to show that s and $V(x, B)$ contain the same set of elements.

First assume C is in s, that is, $C \in visible_T(A, x)$. Since A is live in T, Lemma 6 implies that C is live in T. Then Lemma 16 implies that there exists $D \in anc(C)$ for which $V(x, D)$ is defined and C is an element of $V(x, D)$. Since B is the principal element for x in V, the sequence extension property of the definition of version maps implies that C is also an element of $V(x, B)$.

Conversely, assume that C is an element of $V(x, B)$. Lemma 16 implies that $C \in visible_T(B)$. Condition (d12) for $\mathscr{A}''$ implies that $B \in anc(A)$. Thus $C \in visible_T(A)$.

It is easy to check that property (c) holds, once we know that the definability conditions correspond. Therefore, k' is a possibilities mapping. □

THEOREM 18. *$h \circ h'$ is a simulation of $\mathscr{A}$ by $\mathscr{A}''$.*

PROOF. Immediate from Lemmas 15, 17, and 1. □

8. AN ALGEBRA BASED ON VALUE MAPS

The previous section described a version of a locking algorithm in which considerable information (the sequences of versions) were retained. In

this section, we describe the fourth level of our algorithm. In this level, we optimize the locking algorithm of the previous level by condensing some of the information retained. Namely, it turns out not to be necessary to retain the complete sequences of versions; rather, we can manage by retaining only the latest value of the object for each action.

Note that we can prove a simulation result after eliminating information precisely because possibilities maps are able to yield sets of states rather than single states. The sets of states serve to replace the eliminated information.

8.1. Value Maps

As before, we introduce another data structure. This one records, for each object and action, the latest value of the object which is available to the action.

A *value map* is a partial mapping V from *obj x act* to *values*(*obj*), such that the following properties are satisfied:

- $V(x, U)$ is defined for all x.
- Each $V(x, A) \in values(x)$.
- For each x, if $V(x, A)$ and $V(x, B)$ are both defined, then either $A \in desc(B)$ or $B \in desc(A)$.

If A is the least action for which $V(x, A)$ is defined, then we call A the *principal action* for x in V; in this case, if $V(x, A) = u$, we call u the *principal value* of x in V.

If V is a version map, then let $eval(V)$ be the value map defined on exactly the same domain, so that $eval(V)(x, A) = result(x, V(x, A))$.

LEMMA 19. *Let V be a version map, $x \in obj$. Then the principal action for x in V is the same as the principal action for x in $eval(V)$, and the principal value of x in V is the same as the principal value of x in $eval(V)$.*

PROOF. Straightforward. □

8.2. Definition of the Algebra

We define another algebra $\mathcal{A}''' = \langle A''', \sigma''', \Pi''' \rangle$ as follows. A''' is the set of pairs (T, V), where T is an AAT and V is a value map. σ''' consists of the trivial AAT consisting of a single node U with status 'active', and the value map which has $V(x, U)$ equal to $init(x)$, for all x, and is otherwise undefined. Π''' consists of six events (a)–(f).

In all the events, we assume that $A \in act - \{U\}$. Events (a)–(c), (e), and

(f) are identical to the corresponding events of $\mathcal{A}''$. Event (d) is also identical, except for the change indicated below.

(d2) Effect
(d24) $V(x, A) \leftarrow update(A)(u)$.

8.3. Simulation

Define a mapping h'' from $\mathcal{A}'''$ to $\mathcal{A}''$ as follows. Let $h''(T, V) = \{(T, W): eval(W) = V\}$. h'' maps all events to events of the same name.

LEMMA 20. *h'' is a simulation of $\mathcal{A}''$ by $\mathcal{A}'''$.*

PROOF. It suffices to show that h'' is a possibilities mapping. Properties (a) and (d) are easy to check. Let $\pi' \in \Pi'''$. If π' is any event except for a perform event, then properties (b) and (c) are immediate.

Assume π' is $\mathbf{perform}_{A,u}$. Assume (T, V) is computable in $\mathcal{A}'''$, $(T, W) \in h'''(T, V)$, (T, W) is computable in $\mathcal{A}''$, π' is defined for (T, V), and $(T', V') = \pi'(T, V)$. Lemma 19 implies that property (b) holds, that is, that $\pi = \mathbf{perform}_{A,u}$ is defined on (T, W). It follows from the effects of the two events that $\pi(T, W) = (T', W')$ for some version map W'. In order to show property (c), it suffices to show that $eval(W') = V'$. Since $eval(W) = V$, we need to consider only the values which change because of the present event, that is, we need to show that $result(x, W'(x, A)) = V'(x, A)$. But $result(x, W'(x, A)) = result(x, W(x, B) \circ (A))$, where B is the principal action for x in W, $= update(A)(result(x, W(x, B)))$, $= update(A)(V(x, B))$ since $eval(W) = V$. But B is the principal action for x in V, by Lemma 19, so $u = V(x, B)$. Therefore, the latest term in the extended equality is equal to $update(A)(u)$, which is equal to $V'(x, A)$ by definition. □

THEOREM 21. *$h \circ h' \circ h''$ is a simulation of $\mathcal{A}$ by $\mathcal{A}'''$.*

PROOF. Immediate from Lemmas 18, 20, and 1. □

9. THE ALGORITHM

The only remaining task is to describe a distributed locking algorithm and show that it simulates the previous algorithm. In this section, a slightly simplified version (which doesn't distinguish read and write steps) of Moss' algorithm is described using a distributed algebra.

9.1. Notation and Definitions

Let $[k]$ denote $\{1, \ldots, k\}$.

We fix a particular k as the number of nodes. For convenience, we designate the nodes by identifiers in $[k]$.

Let *home*: $(act - \{U\}) \cup obj \to [k]$, with $home(A) = home(object(A))$ for all $A \in accesses$. Thus *home* partitions the actions and objects among the nodes. Let *origin*: $(act - \{U\}) \to [k]$ be defined so that $origin(A) = home(A)$ if $parent(A) = U$, and $= home(parent(A))$ otherwise.

In order to describe the local state of each node, it is convenient to define a generalization of action trees. Thus we define an *action summary* T to consist of components $vertices_T$, $active_T$, $committed_T$, and $aborted_T$, where $vertices_T$ is any finite subset of act (not necessarily closed under the parent operation) and the remaining three components form a partition of $vertices_T$. The notation $done_T$ and $status_T$ is also extended in the obvious way. If T and T' are action summaries or action trees, we say that $T \leq T'$ provided that $vertices_T \subseteq vertices_{T'}$, and correspondingly for $committed_T$ and $absorbed_T$. We also define $T'' = T \cup T'$ so that $vertices_{T''} = vertices_T \cup vertices_{T'}$, and similarly for $committed_{T''}$ and $aborted_{T''}$. An action summary will be used to describe partial knowedge of the latest status of the transactions.

9.2. Definition of the Algebra

We describe the algorithm as the algebra $\mathcal{B} = \langle B, \tau, P \rangle$, which is distributed over $I = [k] \cup \{\text{'buffer'}\}$. The elements of $[k]$ correspond to k nodes of a distributed system, and the buffer corresponds to the entire message system. The components are defined as follows. Let B be the Cartesian product of state sets B_i, where $i \in I$.

If $i \in [k]$ (that is, if i corresponds to a node), then B_i consists of the values of two variables, $i.T$ which contains an action summary, and $i.V$, which contains a value map. The action summary recorded in $i.T$ represents node i's knowledge of the latest status of various transactions. The value map in $i.V$ contains the latest value map information for all objects whose home is i.

If $i =$ 'buffer', then B_i consists of the values of variables M_i, $j \in [k]$, each of which contains an action summary. The action summary in M_j represents all the information which has been sent to node j during the entire computation.

The initial state τ is a vector of initial states for all the components. If $j \in [k]$, then τ_i has $i.T$ initialized as the trivial action summary, having no vertices, and $i.V$ initialized so that $i.V(x, U) = init(x)$ for all x with

$home(x) = i$, and otherwise undefined. If $i =$ 'buffer', then τ_i has each M_j equal to the trivial action summary.

The algorithm has eight kinds of events. Six correspond closely to the six events of $\mathscr{A}'''$—four record the creation, commit, and abort of actions and the performance of data accesses and two manipulate locks. The other two correspond to the sending and receiving of messages. The events are listed below. As usual, we present them by listing a precondition and the effect on the state. In addition, we define $d(\pi)$, the doer of each step.

In all cases, we assume that $A \in act - \{U\}$;

(a) **create**$_{i,A}$, $origin(A) = i$
  (a1) Precondition
    (a11) $A \notin i.vertices_T$.
    (a12) If $parent(A) \neq U$, then $parent(A) \in i.vertices_T - i.committed_T$.
  (a2) Effect
    (a21) $i.vertices_T \leftarrow i.vertices_T \cup \{A\}$.
    (a22) $i.status_T(A) \leftarrow$ 'active'.
  (a3) Doer: i

(b) **commit**$_{i,A}$, $A \notin accesses$, $home(A) = i$
  (b1) Precondition
    (b11) $A \in i.active_T$.
    (b12) $children(A) \cap i.vertices_T \subseteq i.done_T$.
  (b2) Effect
    (b21) $i.status_T(A) \leftarrow$ 'committed'.
  (b3) Doer: i

(c) **abort**$_{i,A}$, $A \notin accesses$, $home(A) = i$
  (c1) Precondition
    (c11) $A \in i.active_T$.
  (c2) Effect
    (c21) $i.status_T(A) \leftarrow$ 'aborted'.
  (c3) Doer: i

(d) **perform**$_{i,A,u}$, $A \in accesses$, $x = object(A)$, $u \in values(x)$, $home(A) = i$, $home(x) = i$
  (d1) Precondition
    (d11) $A \in i.active_T$.
    (d12) $\{B: i.V(x, B) \text{ is defined}\} \subseteq proper\text{-}anc(A)$.
    (d13) u is the principal value of x in $i.V$.
  (d2) Effect
    (d21) $i.status_T(A) \leftarrow$ 'committed'.
    (d22) $i.V(x, A) \leftarrow update(A)(u)$.

(d3) Doer: i

(e) **release-lock**$_{i,A,x}$, $home(x) = i$

(e1) Precondition

(e11) $i.V(x, A)$ is defined.

(e12) $A \in i.committed_T$.

(e2) Effect

(e21) $i.V(x, parent(A)) \leftarrow i.V(x, A)$.

(e22) $i.V(x, A) \leftarrow$ undefined.

(e3) Doer: i

(f) **lose-lock**$_{i,A,x}$, $home(x) = i$

(f1) Precondition

(f11) $i.V(x, A)$ is defined.

(f12) $anc(A) \cap i.aborted_T \neq \emptyset$.

(f2) Effect

(f21) $i.V(x, A) \leftarrow$ undefined.

(f3) Doer: i

(g) **send**$_{i,j,T'}$, T' an action summary

(g1) Precondition

(g11) $T' \leq i.T$.

(g2) Effect

(g21) $M_j \rightarrow M_j \cup T'$.

(g3) Doer: i

(h) **receive**$_{i,T'}$, T' an action summary

(h1) Precondition

(h11) $T' \leq M_i$.

(h2) Effect

(h21) $i.T \leftarrow i.T \cup T'$.

(h3) Doer: buffer

Thus (a)–(f) correspond closely to (a)–(f) of $\mathscr{A}'''$. Events (g) and (h) are the new communication events. These conditions say that any communication is allowed at any time, which sends any of i's action summary information from i to j.

LEMMA 22. *$\mathscr{B}$ is an algebra, which is distributed over I using d.*

PROOF. Straightforward. □

9.3. Simulation

Now define an interpretation h''' from $\mathscr{B}$ to $\mathscr{A}'''$ by mapping the first six types of events to the events of the same name, suppressing the index in $[k]$, and mapping the other two types of events to Λ.

If $b \in B$, then we add "[b]" to the end of a variable name to denote the value of that variable in state b.

For each $i \in I$, we define a mapping h_i from B to $\mathcal{P}(A''')$ as follows. If $i \in [k]$, then $(T, V) \in h_i(b)$ exactly if (T, V) is computable in $\mathcal{A}'''$ and the following are true:

- $vertices_T \cap \{A: origin(A) = i\} \subseteq i.vertices_T[b] \subseteq vertices_T$.
- $committed_T \cap \{A: home(A) = i\} \subseteq i.committed_T[b] \subseteq committed_T$.
- $aborted_T \cap \{A: home(A) = i\} \subseteq i.aborted_T[b] \subseteq aborted_T$.
- $i.V[b]$ is the restriction of V to $\{(x, A): home(x) = i\}$.

If $i =$ 'buffer', then $(T, V) \in h_i(b)$ exactly if (T, V) is computable in $\mathcal{A}'''$ and $M_j[b] \leq T$ for each $j \in [k]$.

If $(T, V) \in h_i(b)$, then we also say that (T, V) is *i-consistent* with b.

We now proceed to prove lemmas corresponding to the properties required in the definition of a local mapping. The proofs are long, but are very straightforward case analyses.

LEMMA 23. *For all* $i \in \Gamma$, $\sigma''' \in h_i(\tau)$.

PROOF. Immediate from the definitions. □

LEMMA 24. *Assume* $i \in I$. *Assume* $\pi' \in P$, $d(\pi) = i$, $\pi = h'''(\pi') \in \Pi'''$, *a and a' are computable in* $\mathcal{A}'''$ *and* $\mathcal{B}$, *respectively*, $a \in h_i(a')$ *and* $a' \in domain(\pi')$. *Then* $a \in domain(\pi)$.

PROOF. Let a be (T, V).

First, assume that π' is $\mathbf{create}_{i,A}$, so that π is $\mathbf{create}_A$. Then $origin(A) = i$. Since $a' \in domain(\pi')$, $A \notin i.vertices_T[a']$. Since (T, V) is i-consistent with a', $A \notin vertices_T$, thus showing (a11). If $parent(A) = U$, then the fact that (T, V) is computable and Lemma 16 imply that $parent(A) \in active_T$, thus showing (a12) for this case. On the other hand, if $parent(A) \neq U$, then the precondition for π' shows that $parent(A) \in i.vertices_T[a'] - i.committed_T[a']$. The fact that (T, V) is i-consistent with a' implies that $parent(A) \in vertices_T - committed_T$. Thus (a12) holds.

Second, consider $\pi' = \mathbf{commit}_{i,A}$, so that π is $\mathbf{commit}_A$. The precondition for π' shows that $A \in i.active_T[a']$. The fact that (T, V) is i-consistent with a' implies that $A \in active_T$, thus showing (b11). The precondition for π' shows that $children(A) \cap i.vertices_T[a'] \subseteq i.done_T[a']$. The fact that (T, V) is i-consistent with a' implies that $children(A) \cap vertices_T \subseteq done_T$, thus showing (b12).

Third, assume $\pi' = \mathbf{abort}_{i,A}$, so that π is $\mathbf{abort}_A$. This case is similar to the first half of the previous case.

Fourth, assume $\pi' = \mathbf{perform}_{i,A,u}$, so that π is $\mathbf{perform}_{A,u}$. Then $home(A) = i$. Assume $object(A) = x$, so that $home(x) = i$. (d11) is argued as in the preceding two cases. We show (d12). Choose B so that $V(x, B)$ is defined. Since (T, V) is i-consistent with a' and $home(x) = i$, $i.V(x, B)[a']$ is also defined. The precondition for π' implies that $B \in proper\text{-}anc(A)$, as needed. Next, we show (d13). The precondition for π' implies that u is the principal value for x in $i.V[a']$. Since (T, V) is i-consistent with a', u is also the principal value for x in V, as needed.

If π' is one of (e) and (f), then π' involves some x with $home(x) = i$. Assume that π' involves A. The precondition for π' implies that $i.V(x, A)[a']$ is defined. Since (T, V) is i-consistent with a', it follows that $V(x, A)$ is defined, thus showing both (e11) and (f11).

If π' is a $\mathbf{release\text{-}lock}_{i,A,x}$ step, then the precondition for π' implies that $A \in i.committed_T[a']$. Since (T, V) is i-consistent with a'. $A \in committed_T$, thus showing (e12).

Finally, if π' is a $\mathbf{lose\text{-}lock}_{i,A,x}$ step, the precondition for π' implies that $anc(A) \cap i.aborted_T[a'] \neq \emptyset$. Since (T, V) is i-consistent with a', it follows that A is dead in T, thus showing (f12). □

LEMMA 25. *Assume* $i, j \in I$. *Assume* $\pi' \in P$, $d(\pi') = i$, $\pi = h'''(\pi') \in OP'''$, *a and* a' *are computable in* $\mathcal{A}'''$ *and* $\mathcal{B}$, *respectively*, $a \in h_i(a') \cap h_j(a')$, *and* $a' \in domain(\pi')$. *If* $b' = \pi'(a')$, *then* $\pi(a) \in h_j(b')$.

PROOF. Let $a = (T, V)$ and $\pi(a) = (T', V')$. Lemma 24 implies that $a \in domain(\pi)$.

If $j \neq i$, then it is easy to see that all the containments are preserved, since the sets of actions on the right sides are only increased, while the sets on the left sides are unchanged. The property involving V is also easily seen to be preserved. So assume $j = i$. We consider the six kinds of events in turn.

First, assume π' is of the form $\mathbf{create}_{i,A}$, $\mathbf{commit}_{i,A}$, or $\mathbf{abort}_{i,A}$. Then $V' = V$, and T' is exactly like T except that A is added to $vertices_T$, $committed_T$, or $aborted_T$ as appropriate. Also, b' is just like a' except that A is added to $i.vertices_T$, $i.committed_T$, or $i.aborted_T$. as appropriate. Since (T, V) is i-consistent with a', it is easy to see that all the containments change in such a way as to insure that (T', V') is i-consistent with b'.

If π' is of the form $\mathbf{perform}_{i,A,u}$, then $home(A) = i$. Let $x = object(A)$. Then $home(X) = i$. T' is just like T except that A is added to $committed_T$ and is given label u, and $data_T$ is augmented with all pairs in $\{(B, A): B \in datasteps_T(x)\} \cup (A, A)$. V' is just like V except that $V'(x, A)$ is defined to be $update(A)(u)$. b' is just like a' except that A is added to $i.committed_T$, and $i.V(x, A)$ is defined to be $update(A)(u)$.

Since (T, V) is i-consistent with a', it is easy to see that (T', V') is i-consistent with b': most of the properties are immediate. We just check the last property; the only change involves A. We have already noted that $i.V(x, A)[b'] = update(A)(u) = V'(x, A)$. This is as needed.

If π' is of one of the forms (e) or (f), then $T' = T$ and $i.T(b') = i.T[a']$. Thus it is clear that the containments are all preserved. It is also easy to check that the final property is preserved. □

LEMMA 26. *Assume $i, j \in I$. Assume $\pi' \in P$, $d(\pi') = i$, $h(\pi') = \Lambda$, and a' are computable in $\mathscr{A}'''$ and $\mathscr{B}$, respectively, $a \in h_i(a') \cap h_j(a')$, and $a' \in domain(\pi')$. If $b' = \pi'(a')$, then $a \in h_j(b')$.*

PROOF. Let $a = (T, V)$.

First, assume that π' is $\mathbf{send}_{i,i',T'}$. If $j \neq$ 'buffer', then $b'_j = a'_j$, and the conclusion is immediate. So assume that $j =$ 'buffer'. Since (T, V) is j-consistent with a', each action summary $M_l[a'] \leq T$. The precondition for π' implies that $T' \leq i.T[a']$. Since (T, V) is i-consistent with a', it follows that $i.T[a'] \leq T$, and hence $T' \leq T$. Now, each $M_l[b'] \leq M_l[A'] \cup T'$. Therefore, each $M_l[b'] \leq T$, as needed.

Next, assume that π' is of the form $\mathbf{receive}_{i',T'}$, so that $i =$ 'buffer'. The only nontrivial case is $j = i'$. We must show that $j.T[b'] \leq T$. But $j.T[b'] = j.T[a'] \cup T'$. The j-consistency of (T, V) with a' shows that $j.T[a'] \leq T$. The precondition for π' shows that $T' \leq M_j[a']$. Since (T, V) is j-consistent with a', $M_j[a'] \leq T$. Thus $T' \leq T$. Therefore, $j.T[b'] \leq T$, as needed. □

LEMMA 27. *h''' and h_i, $i \in I$, form a local mapping from $\mathscr{B}$ to $\mathscr{A}'''$.*

PROOF. Immediate from Lemmas 23, 24, 25, and 26. □

Now extend h''' to $B \cup P$, by defining $h'''(b) = \bigcap_{i \in I} h_i(b)$.

LEMMA 28. *h''' is a simulation of $\mathscr{A}'''$ by $\mathscr{B}$.*

PROOF. Immediate by Lemmas 27 and 4. □

The main correctness theorem now follows.

THEOREM 29. *The mapping $h \circ h' \circ h'' \circ h'''$ is a simulation of $\mathscr{A}$ by $\mathscr{B}$.*

PROOF. Immediate from Lemma 28, Lemma 1, and Theorem 21. □

10. CONCLUSIONS

In this paper, we have presented a detailed proof of a variant of Moss' concurrency control algorithm for nexted transactions. Along the way, we have developed a substantial amount of basic theory for nested transactions. The basic framework, especially the definitions and results involving visibility, should be of further use.

There is much more to be done, however. The framework presented in this paper is not powerful enough to describe all the correctness conditions one might want for nested transactions. In particular, we do not model the correspondence between what the system does and what it is requested to do by the transactions. This deficiency is at least partly due to the fact that we have chosen not to model the transactions explicitly. In order to describe everything we might want, we will probably have to incorporate some type of model for the transactions into the framework.

We have only proved correctness of one variant of Moss' algorithm. There are many other related algorithms for which similar proofs ought to be developed. Certainly, Moss' complete algorithm (with a distinction between read and write operations) should be proved correct; we do not expect this extension to be very difficult. The orphan algorithm mentioned in the introduction should be verified; obtaining an understandable proof for this algorithm seems like a much harder task. Also, other implementations for nested transactions, such as Reed's, should be proved correct. It would be interesting to see to what extent the theory developed for one of these algorithms is usable for the others.

The proof presented here has a very interesting structure. It describes algorithms as algebras and uses a series of five levels of abstraction. Correctness is shown using four simulation mappings. The interesting and nontrivial concurrency control arguments are made in proving the correctness of the first two simulations. The correctness of the first simulation expresses the fact that certain conditions imply serializability. The correctness of the second simulation expresses the fact that a form of locking satisfies these conditions. Successive levels refine the algorithm, providing more implementation detail, condensing the information that is kept, and distributing the processing. Proofs at these lower levels are straightforward checks of the local mapping properties.

There is more to be done in exploring the usefulness of this proof structure for other distributed algorithms.

ACKNOWLEDGMENTS

Many other people have contributed their ideas and efforts to this work. Barbara Liskov suggested formal treatment of this area and monitored proposed for-

malizations for their faithfulness in representing the behavior of the Argus system. John Goree used a much earlier draft of the current paper as a starting point for the work in his Master's thesis; in the process of writing his thesis, he discovered several major ways of clarifying the ideas of this paper. Some of the ideas Gene Stark developed for his thesis [11] have found their way into the present paper. Bill Weihl and Gene Stark contributed helpful criticisms of the early drafts. Paris Kanellakis and two anonymous referees contributed many very helpful suggestions for the presentation.

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 Contracts DAAG-29-79-C-0155 and DAAG29-84-K-0058, by the National Science Foundation under Grants MCS-79-24370 and DCR-83-02391, and by the Defense Advanced Research Projects Agency (DARPA) under Grants N00014-76-C-0944 and N00014-83-K-0125.

REFERENCES

1. Berri C, Bernstein PA, Goodman N, Lai MY, Shasha DE: A concurrency control theory for nested transactions. *Proc 2nd ACM Symp on Principles of Distributed Computing*, pp 45–62, 1983.
2. Bernstein P, Goodman N: Concurrency control algorithms for multiversion database systems. *Proc ACM SIGACT-SIGOPS Symp on Principles of Distributed Computing*, pp 209–215, 1982.
3. Eswaren KP, Gray JN, Lorie RA, Traiger IL: The notions of consistency and predicate locks in a database system. *Comm ACM* 19, 1976.
4. Goree J: Internal consistency of a distributed transaction system with orphan detection. *Technical Report MIT/LCS/TR-286*, MIT Laboratory for Computer Sci, Cambridge, MA, January, 1983.
5. Kanellakis P, Papadimitriou C: On concurrency control by multiple versions. *Proc ACM Symp on Principles of Database Systems*, pp. 76–82, 1982.
6. Lamport L: Time, clocks and the ordering of events in a distributed system. *Comm ACM* 21, 1978.
7. Liskov B, Scheifler R: Guardians and actions: Linguistic support for robust, distributed programs. *Proc 9th ACM SIGACT-SIGPLAN Symp on Principles of Programming Languages*, pp 7–19, 1982.
8. Moss JEB: *An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, MA, 1985.
9. Randell B: System structures for software fault tolerance. *Proc Int Conf on Reliable Software, 1975*; *SIGPLAN Notices* 10:437–457; also in: *IEEE Trans Software Eng* 1:220–232, 1975.
10. Reed DP: Implementing atomic actions on decentralized data. *ACM Trans Comp Systems* 1:3–23, 1983.
11. Stark E: Foundations of a theory of specification for distributed systems. Ph.D Thesis, MIT Laboratory for Computer Science, Cambridge, MA, August, 1984.