Concurrency Control for Resilient Nested Transactions

Nancy A Lynch
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

January, 1983

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) 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 insure that all executions of transactions be serializable

*This work was supported in part by the Army Research Office under Contract # DAAG 29 79 C 0155, Defense Advanced Research Projects Agency # N00014 75 C0661, and by the National Science Foundation under Grants MCS 79 24370

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 [EGLT], and those based on timestamps [La] 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 [Re, M, LiS] 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) sub transactions The intention is that the sub transactions 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 [Ra] 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 is currently implementing a system which uses multiple versions of data to implement nested transactions which tolerate failures of sub transactions Moss has abstracted away from Reed's specific implementation of nested transactions, and has presented a clear intuitive 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

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 sub transactions 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 insure 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 careful correctness proof is given

The style of correctness proof for the algorithm appears to be quite interesting in its own right The description of the algorithm is presented in a series of levels, each of which is an "event state" algebra with unary operations, and each (but the first) of which "simulates" the previous one The basic problem statement is given as the highest level algebra, and successively lower levels provide increasing amounts of implementation detail In particular, both the problem specification and the implementation are presented as the same kind of mathematical object, an event state algebra At every level, we want to present algorithms with the maximum possible amount of nondeterminism consistent with correctness, not forcing any unnecessary implementation decisions Therefore, we do not describe algorithms in the usual way, using programs with specified flow of control Rather, we present algorithms as collections of events with corresponding preconditions

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 more "concrete" algebra sometimes to contain less information than the more "abstract" algebra For example, it might be easy to prove correctness of an algorithm which maintains lots of auxiliary information The correctness of an algorithm which maintains less information could be proved, in our model, by showing that it simulates the algorithm which maintains the auxiliary information

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" In a distributed algebra, the state set is just a Cartesian product, with event preconditions and transitions 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 [G] presents a more complete (and slightly more general) development of the technique than is presented in this paper

The 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 [G] has given 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

Other related work is that of Stark [S] He is carrying out a very general treatment of event state algebras, incorporating considerations of modularity to a much greater extent than is present in this paper, and handling fairness and eventuality properties as well as safety properties

2 Event-State Algebras

In this section, we describe the event state algebra framework

An event state algebra $\mathcal{A} = \langle A, \sigma, \Pi \rangle$, consists of a set A of states, an element $\sigma \in A$, the initial state, and a set Π of partial unary operations In this paper, we will usually refer to an event state algebra as simply an algebra

Let a be a state, and let $\Phi = (\pi_1, \ldots, \pi_k)$ be any finite sequence of operations chosen from OP Then Φ is said to be valid from a provided $b = \pi_k(\pi_{k-1}(\ldots(\pi_1(a))\ldots))$ is defined, in this case, b is called the result of Φ applied to a An infinite sequence of operations 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 $\sigma \vdash b$

Now assume $\mathcal{A} = \langle A, \sigma, \Pi \rangle$ and $\mathcal{A}' = \langle A', \sigma', \Pi' \rangle$ are two algebras An interpretation of $\mathcal{A}$ by $\mathcal{A}'$ is a mapping $h: \Pi' \rightarrow \Pi \cup \{\Lambda\}$ We extend h to map operation sequences of $\mathcal{A}'$ to operation sequences of $\mathcal{A}$ in the obvious way (deleting occurrences of Λ) An interpretation, h, is a simulation of $\mathcal{A}$ by $\mathcal{A}'$ provided that $h(\Phi')$ is a valid operation sequence for $\mathcal{A}$ whenever Φ' is a valid operation sequence for $\mathcal{A}'$

Lemma 1 Assume that $\mathcal{A}$, $\mathcal{A}'$ and $\mathcal{A}''$ are algebras, that h is a simulation of $\mathcal{A}$ by $\mathcal{A}'$ and h' is a simulation of $\mathcal{A}'$ by $\mathcal{A}''$ Then $h \circ h'$ is a simulation of $\mathcal{A}$ by $\mathcal{A}''$

Proof Straightforward

□

Next, we give a sufficient condition for a mapping h to be a simulation Let $h: A' \cup \Pi' \rightarrow \mathcal{P}(A) \cup \Pi \cup \{\Lambda\}$ be such that $h(a') \in \mathcal{P}(A)$ for all $a' \in A$, and h restricted to Π' is an interpretation Then h is a possibilities mapping from $\mathcal{A}'$ to $\mathcal{A}$ provided the following are true

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

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

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

(c) If $h(\pi') = \Lambda$, then $a \in h(b')$

Lemma 2 Let h be a possibilities mapping from $\mathcal{A}'$ to $\mathcal{A}$ If Φ' is a valid operation sequence for $\mathcal{A}'$, and $h(\Phi') = \Phi$, then Φ is a valid operation 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

□

If we think of $\mathcal{A}'$ as a "concrete" algebra, and $\mathcal{A}$ 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 just single abstract states

An algebra, $\mathcal{A} = \langle A, \sigma, \Pi \rangle$, 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 operation, and the following two conditions are satisfied

(Local Domain) Let $i = d(\pi)$ If $a, b \in A$ and $a_i = b_i$, then $a \in domain(\pi)$ if and only if $b \in domain(\pi)$

(Local Changes) If $a, b \in domain(\pi)$, $a' = \pi(a)$, $b' = \pi(b)$ and $a_i = b_i$, then $a'_i = b'_i$

We now consider the simulation of an algebra by a distributed 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 node can tell 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 just those which 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 any algebra Let h be an interpretation from $\mathcal{A}'$ to $\mathcal{A}$ For each $i \in I$, let h_i $A' \rightarrow \mathcal{P}(A)$ be such that h_i depends on A'_i only i e 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 local mapping from $\mathcal{A}'$ to $\mathcal{A}$ provided the following conditions are satisfied

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

Assume $\pi' \in \Pi'$, $d(\pi) = i$ Assume a and a' are computable in $\mathcal{A}$ and $\mathcal{A}'$, respectively Assume $a \in h_i(a')$ Assume $a' \in domain(\pi')$, and $b' = \pi'(a')$

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

(c) Assume $h(\pi') = \pi \in \Pi$, $j \in I$ and $a \in h_j(a')$ Then $\pi(a) \in h_j(b')$

(d) Assume $h(\pi') = \Lambda$, $j \in I$ and $a \in h_j(a')$, Then $a \in h_j(b')$

Lemma 4 Let $\mathcal{A}$ and $\mathcal{A}' = \langle A', \sigma', \Pi' \rangle$ be algebras, where $\mathcal{A}'$ is distributed over I Assume that h and h_i, $i \in I$ form a local mapping from $\mathcal{A}'$ to $\mathcal{A}$ Extend h to $A' \cup \Pi'$ by defining $h(a') = \bigcap_{i \in I} h_i(a')$ Then h is a possibilities mapping from $\mathcal{A}'$ to $\mathcal{A}$

Proof We check the three 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) and (c) 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 $a \in domain(\pi)$ 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

(c) 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 operations of a more concrete algebra to be interpreted by sequences of operations of a more abstract algebra (See Goree (G) for definitions and uses for this generalization) Also, sets of initial states rather than single initial states are probably useful

3 Action Trees

In this section, basic definitions are given for action trees and serializability

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 $x \in obj$

Let act be a universal set of actions (i e transactions) 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 $\{(A,B) \in act^2 \mid parent(A) = parent(B)\}$ If $A \in act$, let children(A) denote $\{B \in act \mid parent(B) = A\}$ 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 desc(A) (resp proper anc(A)) be the set of proper descendants (resp ancestors) of A

It might be convenient for the reader to think of this 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

Let seq $\subseteq$ siblings be any fixed partial order, representing sequential dependency If (A,B) $\in$ seq, it means that A is constrained to run before B For the sake of notational simplicity, we are assuming this relation is also fixed a priori, this amount to assuming that the "name" of any action carries within it information about which siblings the action can assume have completed The use of an arbitrary partial order is a generalization of both the total order usually specified for the steps which occur within a single level transaction, and the unconstrained order usually specified among the transactions themselves We also assume a priori determination of which actions actually access data, which objects they access and the functions they perform on those objects let accesses denote the leaves of the tree described above (We assume that U $\notin$ accesses, so that the 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 Let sameobject denote {(A,B) $\in$ accesses ^{2} object(A) = object(B)}

I am departing from the usual approach in serializability theory by including a particular function (rather than an uninterpreted function) in the definition of an action which accesses data This is because I want to state correctness conditions in terms of preserving certain relationships among the data values seen and written This 'semantic" style of correctness condition seems to me to be more basic than the usual correctness definitions in serializability theory, in that it says less to constrain the implementation

Note that the usual read and write operations of serializability theory can be regarded as special cases of my 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

Next, I 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

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 non access 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)

$label_T$ $datasteps_T$ $\rightarrow$ 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)

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, object(B) = x} Let seq_T denote seq $\cap$ $(vertices_T)^2$

Next, we 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 desc(lca(A,B)) $\subseteq$ $committed_T$} That is, $visible_T$(A) is just the set of actions whose existence is known to action A, because they and all their ancestors, up to and not including some ancestor of A, have committed 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 A $\in$ desc(B), then B $\in$ $visible_T$(A)

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 $desc(lca(A,C))$ We must show that $D \in committed$ If $D \in$ proper $desc(lca(A,B))$, then the fact that $A \in visible_T(B)$ implies the result So assume that $D \notin$ proper $desc(lca(A,B))$ 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 $desc(lca(B,C))$, 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

□

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 $desc(lca(A,B))$, since $B \in visible_T(A)$ Thus, $C \in anc(lca(A,B)) \subseteq anc(A)$, so A is dead in T, a contradiction

□

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 $x_i = result(x,s')$ 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 $\leq$

Let T be an action tree A partial order $p \subseteq$ siblings is linearizing for T provided p totally orders all siblings in T A linearizing partial order p induces a total order, $induced_{T,p}$, on $datasteps_T$, in the obvious way If $A \in datasteps_T(x)$ and p is a linearizing partial order for T, let $preds_{T,p}(A)$ denote $\langle\langle \{B \in visible_T(A,x)\ (B,A) \in induced_{T,p}$ and $B \neq A\}, induced_{T,p} \rangle\rangle$

A linearizing partial order p for T is said to be a serializing partial order for T provided p is consistent with seq, and $label_T(A) = result(x, preds_{T,p}(A))$ for all $A \in datasteps_T(x)$ T is said to be serializable provided there exists some serializing partial order for T

Stating the simplest correctness requirements only requires consideration of actions whose effects become 'permanent" Therefore, we restrict attention to a portion of T, as follows 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)

If $A \in vertices_{perm(T)}$, then $status_{perm(T)}(A) = status_T(A)$

If $A \in datasteps_{perm(T)}$, then $label_{perm(T)}(A) = label_T(A)$

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)

□

We will require that any tree T created by our algorithm have perm(T) serializable

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 (KP, BG) 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 to me, 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 implementations as yet unforeseen

4 An Algebra Based on Action Trees

We now define a set of operations on action trees That is, we define an algebra $\mathcal{A} = \langle A, \sigma, \Pi \rangle$, 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 operations described in (a) (d) below We define the operations 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 constrain the ranges of all of the operations to be subsets of 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 operation on T

In all operations, we assume that $A \in act \{U\}$

(a) **create**$_A$

(a1) Precondition
(a11) $A \notin vertices_T$
(a12) $parent(A) \in vertices_T$ $committed_T$
(a13) If $(B,A) \in seq$ and $B \neq A$, then $B \in done_T$

(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**$_{A,u}$, $A \in accesses$, $x = object(A)$, $u \in values(x)$

(d1) Precondition
(d11) $A \in active_T$

(d2) Effect
(d21) $status_T(A) \leftarrow$ 'committed'
(d22) $label_T(A) \leftarrow u$

5 Augmented Action Trees

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

In this section, a new structure called an "augmented action tree" is defined Augmented action trees are just action trees with a little additional information Namely, in the spirit of the earlier definitions, some information is added which describes a sequence of versions for each data object Serializability is defined for augmented 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 Thus, this structure can be useful in proofs of serializability for action trees

An augmented action tree (AAT), T, is a pair (S,D), where S is an action tree and $D \subseteq sameobject_S$ is a partial order on $datasteps_S$ which totally orders the datasteps for each object In this case, we write $\underline{data}_T$ for D We extend action tree notation to T, for example, we write $\underline{datasteps}_T$ to denote $datasteps_S$ If T is an AAT then let $\underline{sibling\ data}_T$ denote $\{(A,B) \in siblings_S\ (C,D) \in data_T$ for some $C \in desc(A)$, $D \in desc(B)\}$ If $A \in datasteps_T(x)$ then let $\underline{v\ data}_T(A)$ denote $\{B \in visible_T(A,x)\ (B,A) \in data_T$ and $B \neq A\}$

The following is a technical lemma needed for the characterization theorem

Lemma 8 Let T be an AAT If there is a cycle of length greater than one in seq $\cup$ sibling $data_T$ then there is a cycle of length greater than one in $seq_T \cup$

sibling $data_T$

Proof Assume not Choose a cycle, C, of minimum length greater than one, in seq U sibling $data_T$ There must be an action, A, on C with A ∉ $vertices_1$ Let (B,A) and (A,C) be the two pairs on C involving A Then both pairs are elements of seq Thus, (B,C) ∈ seq and B ≠ C since seq is a partial order Removing A from C leaves a cycle with at least two elements (B and C), having one fewer element than C This contradicts the minimality of C

□

If T = (S,D) is an AAT, then erase(T) is just the action tree S We extend the definitions of visible, live, dead, linearizing, induced, preds and serializable to an AAT, T, by applying them to erase(T) 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 $data_T$ Data serializability for AAT's provides a sufficient condition for correctness

Lemma 9 Let T be an AAT Let p be a linearizing partial order for T, x ∈ obj, and A ∈ $datasteps_T(x)$ Assume that $induced_{T,p}$ is consistent with $data_T$ Then $preds_{T,p}(A)$ = <<v $data_T(A)$, $data_T$>>

Proof Straightforward

□

Data serializability for AAT's has a cycle free characterization 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 ∈ obj, and every A ∈ $datasteps_T(x)$, it is the case that $label_T(A)$ = $result(x,s)$, where s = <<v $data_T(A)$, $data_T$>>

Theorem 10 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 seq_T U sibling $data_T$

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 ∈ $datasteps_T(x)$, s = <<v $data_T(A)$, $data_T$>> Then $label_T(A)$ = $result(x,preds_{T,p}(A))$, by the definition of serializability, = result(x,s), by Lemma 9

b seq_T U sibling $data_T$ ⊆ p Thus, there are no cycles of length greater than one in seq_T U sibling $data_T$

Now assume a and b Lemma 8 implies that there are no cycles of length greater than one in seq U sibling $data_T$ Let p be any partial order which totally orders all siblings and is consistent with seq U sibling $data_T$ 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 ∈ obj, A ∈ $datasteps_T(x)$ 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 = <<v $data_T$, $data_T$>> Then Lemma 9 implies that s = $preds_{T,p}(A)$, as needed

□

6 An Algebra Based on Augmented Action Trees

In order to prove that an algorithm generates only correct operation sequences, it is helpful to include the additional information present in AAT's Thus, we define operations on AAT's, analogously to the definitions for action trees Once again, we carry out the definitions within the algebra framework defined earlier We define a new algebra $\mathcal{A}'$ = <A , σ', Π'>, where A is the set of AAT's, σ' is the trivial AAT which has a single vertex U with status 'active', and the operations in Π' correspond closely to the operations of $\mathcal{A}$, 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 ∈ $datasteps_T(x)$, B live in T
    Then B ∈ $visible_T(A,x)$
  (d13) If A is live in T, then u = result(x,s),
    where s = <<$visible_T(A,x)$, $data_T$>>

(d2) Effect
  (d23) $data_T$ ← $data_T$ U {(B,A) B ∈ $datasteps_T(x)$} U {(A,A)}

Lemma 11 If T is computable in $\mathcal{A}'$, then the following are true

a If A $\in$ $vertices_T$ and parent(A) $\in$ $committed_T$, then A $\in$ $done_T$

b If A $\in$ $vertices_T$ and (B,A) $\in$ seq and B $\neq$ A, then B $\in$ $done_T$

c U $\in$ $active_T$

d If (B,A) $\in$ $data_T$, then either B is dead in T, or else B $\in$ $visible_T$(A)

e If A $\in$ $committed_T$ and B $\in$ desc(A) $\cap$ $vertices_T$ then either B is dead in T or else B $\in$ $visible_T$(A)

Proof Most of the arguments are straightforward We argue cases d and e

d 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 $perform_{A,u}$ 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)

e 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

□

Lemma 12 Let T and T' be computable in $\mathcal{A}'$, and assume that T $\vdash$ T

a $vertices_T$ $\subseteq$ $vertices_{T'}$, $committed_T$ $\subseteq$ $committed_{T'}$, $aborted_T$ $\subseteq$ $aborted_{T'}$, and $data_T$ $\subseteq$ $data_{T'}$

b If A $\in$ $datasteps_T$ 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 $\in$ $done_T$

□

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

Lemma 13 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 = <<v $data_{perm(T)}$(B), $data_{perm(T)}$>> A is inserted into the tree by a $\mathbf{perform}_{A,u}$ step π, so let the operation 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' = <<$visible_T$ (A,x) $data_T$>> By Lemma 12b 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 12c, 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 11), 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 12e 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 12a, (B,A) $\in$ $data_T$ By Lemma 12d, 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 14 If T is computable in $\mathcal{A}'$, then there are no nontrivial cycles in $seq_{perm(T)} \cup$ sibling $data_{perm(T)}$

Proof Assume the contrary let $(\sigma, A_1, \ldots, A_k = \sigma)$, $k \geq 2$, be a minimum length cycle such that $(A_i, A_{i+1}) \in seq_{perm(T)} \cup$ sibling $data_{perm(T)}$ for all i, $0 \leq i \leq k-1$ Let a sequence Φ of operations 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 Γ is the result of Ψ_i then $A_i \in done_\Gamma$, and $A_{i+1} \notin done_\Gamma$ 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 12a implies that $A_{i+1} \in done_{T'}$, which is a contradiction

Fix i If $(A_i, A_{i+1}) \in seq_{perm(T)}$, then Φ has a prefix $\Psi\pi$, where π is a $\mathbf{create}_{A_{i+1}}$ operation Let T' be the result of Ψ The preconditions for π imply that $A_i \in done_{T'}$ Thus $\Psi_i = \Psi$ suffices

Now assume that $(A_i, A_{i+1}) \in$ sibling $data_{perm(T)}$ 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 $desc(U) \subseteq committed_T$ Now, Φ has a prefix $\Psi\pi$, where π is a $\mathbf{perform}_{C,u}$ step Let T' be the result of Ψ, and T'' the result of $\Psi\pi$ Lemma 12c implies that $(B,C) \in data_{T'}$, so that $B \in datasteps_{T'}$ Since B is live in T (using Lemma 11c), Lemma 12e 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 $desc(lca(B,C)) \subseteq committed_{T'} \subseteq done_{T'}$ 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 $desc(lca(B,C)) \subseteq committed_{T'}$ so $D \in committed_{T'}$ Since $C \notin vertices_{T'}$, 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 12f This means $C \notin vertices_{T'}$ This is a contradiction

□

Theorem 15 If T is computable in $\mathcal{A}'$, then perm(T) is data serializable

Proof Immediate from Lemma 13, Lemma 14 and Theorem 10

□

Next, we show that it is sufficient to restrict attention to correctness of operation sequences for AAT's We define a mapping h from $\mathcal{A}'$ to $\mathcal{A}$ as follows If T is an AAT then $h(T) = \{erase(T)\}$ If π is in Π', then $h(\pi)$ is just the operation in Π with the same name

Lemma 16 h is a simulation of $\mathcal{A}$ by $\mathcal{A}'$

Proof (a) and (c) are immediate To see (b), the first conclusion follows immediately from the fact that $a \in domain(\pi')$ (since only additional constraints are added for $\mathcal{A}'$), note that Theorem 15 implies that the C constraint is always satisfied The second conclusion 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 this section, we introduce another data structure This one records, for each object and action, the sequence of accesses to the object whose result is available to the action

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)

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

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 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 operations defined below in (a) (f)

In all the operations to follow we assume that A ∈ act − {U}

Operations (a) (c) are identical to (a) (c) of $\mathcal{A}'$

(d) **perform**$_{A,u}$, A ∈ accesses, x = object(A), u ∈ values(x)

(d1) Precondition
(d11) A ∈ active$_T$
(d12) {B | V(x,B) is defined} ⊆ proper anc(A)
(d13) u is the principal value of x in V

(d2) Effect
(d21) status$_T$(A) ← 'committed'
(d22) label$_T$(A) ← u
(d23) data$_T$ ← data$_T$ ∪ {(B,A) | B ∈ accesses$_T$(x)} ∪ {(A,A)}
(d24) V(x,A) ← V(x,B) ∘ (A)

(e) **release lock**$_{A,x}$, x ∈ obj

(e1) Precondition
(e11) V(x,A) is defined
(e12) A ∈ committed$_T$

(e2) Effect
(e21) V(x,parent(A)) ← V(x,A)
(e22) V(x,A) ← undefined

(f) **lose lock**$_{A,x}$, x ∈ obj

(f1) Precondition
(f11) V(x,A) is defined
(f12) A is dead in T

(f2) Effect
(f21) V(x,A) ← undefined

Lemma 17 If (T,V) is computable in $\mathcal{A}''$, then the following are true

a If V(x,A) is defined, then A ∈ vertices$_T$

b If B ∈ datasteps$_T$(x) and B is live in T, then there exists A ∈ 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 operation **perform**$_{B,u}$ occurs, we see that V(x,B) is defined, and B ∈ V(x,B) Assume inductively that there is some ancestor C, of B with V(x,C) defined and B ∈ V(x,C) Since B remains live, there are no steps of the form **lose 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 **release 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 ∈ 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

□

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

Lemma 18 h' is a simulation of $\mathcal{A}'$ by $\mathcal{A}''$

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

Condition (d11) for $\mathcal{A}'$ follow immediately from the corresponding condition for $\mathcal{A}''$ We consider (d12) Let B ∈ datasteps$_T$(x), and assume that B is live in T Since (T,V) is computable in $\mathcal{A}''$, Lemma 17 implies that there is some C ∈ anc(B) for which V(x,C) is defined and for which B is an element of V(x,C) Then Lemma 17 implies that B ∈ visible$_T$(C) Since π' is defined on (T,V), (d12) for $\mathcal{A}''$ implies that C ∈ anc(A) Since A ∈ vertices$_T$ Lemma 5 implies that B ∈ visible$_T$(A), as needed

Next, we consider (d13) Assume A is live in T, and let s = <<visible$_T$(A,x), data$_T$>> We must show that u = result(x,s) Let B be the principal action for x in V Condition (d13) for $\mathcal{A}''$ 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 17), it suffices to show that s and V(x,B) contain the same set of elements

First assume C is in s, i e C ∈ visible$_T$(A,x) Since A is live in T, Lemma 6 implies that C is live in T Then Lemma 17 implies that there exists D ∈ 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 17 implies that C ∈ visible$_T$(B) Condition (d12) for $\mathcal{A}''$ implies that B ∈ anc(A) Thus, C ∈ visible$_T$(A)

It is easy to check that the changes correspond correctly, once we know that the definability conditions correspond Therefore, h' is a possibilities mapping

□

Theorem 19 h ∘ h' is a simulation of $\mathcal{A}$ by $\mathcal{A}''$

Proof Immediate from Lemmas 16, 18 and 1

□

8 An Algebra Based on Value Maps

In this section, 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) ∈ values(x), and

for each x, if V(x A) and V(x,B) are both defined, then either A ∈ desc(B) or B ∈ 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

We define another algebra, $\mathcal{A}'''$ = <A''', σ''', Π'''>, 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 the six operations defined below in (a) (f)

In all the operations to follow, we assume that A ∈ act {U} Operations (a) (c), (e) and (f) are identical to the corresponding operations of $\mathcal{A}''$ Operation (d) is also identical, except for the change indicated below

(d2) Effect
(d24) V(x,A) ← update(A)(u)

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 20 Let V be a version map, x ∈ 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

□

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

Lemma 21 h ' is a simulation of $\mathcal{A}''$ by $\mathcal{A}'''$

Proof It suffices to show that h'' is a possibilities mapping The first and last properties are easy to check We consider the second property Let π' ∈ Π'' If π' is one of (a) (c), (e) or (f), then the second property is obvious

Assume π' is **perform**$_{A,u}$ Assume (T,V) is computable in $\mathcal{A}$, (T W) ∈ π (T,V) (T,W) is computable in $\mathcal{A}'$ π' is defined for (T V) and (T ,V') = π (T V) Lemma 20 implies that the definability condition holds, i e that π = **perform**$_{A,u}$ is defined on (T,W) It follows from the effects of the two operations that π(T,W) = (T W') for some version map W' It suffices to show that eval(W') = V Since eval(W) = V, we only need to consider the values which change because of the present operation i e we need to show that result(x W'(x,A)) = V (x,A) But result(x,W'(x A)) = result(x,W(x B) ∘ (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 γ in V, by Lemma 20, 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 22 h ∘ h' ∘ h'' is a simulation of $\mathcal{A}$ by $\mathcal{A}'''$

Proof Immediate from Lemmas 19, 21 and 1

□

9 The Algorithm

A slightly simplified version (which doesn't distinguish read and write steps) of Moss' algorithm is described using a distributed algebra

Let [k] denote {1, ,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}) ∪ obj → [k], with home(A) = home(object(A)) for all A ∈ accesses Thus, home partitions the actions and objects among the nodes Let origin (act {U}) → [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 $\text{vertices}_T \subseteq \text{vertices}_{T'}$, and correspondingly for committed_T and aborted_T We also define T'' = T ∪ T' so that $\text{vertices}_{T''} = \text{vertices}_T \cup \text{vertices}_{T'}$, and similarly for $\text{committed}_{T''}$ and $\text{aborted}_{T''}$

We describe the algorithm as yet another algebra, $\mathfrak{B} = \langle B, \tau, P \rangle$, which is distributed over I = [k] ∪ {'buffer'} The components are defined as follows B is the Cartesian product of B_i, where i ∈ I If i ∈ [k], then B_i consists of the values of variables i T which can contain an action summary, and i V, which can contain a value map defined only for pairs (x,A) having home(x) = i If i = 'buffer' then B_i consists of the values of variables M_j, j ∈ [k], each of which can contain an action summary (The contents of M_j are intended to denote information which has been sent to node j)

τ is a vector of initial states for all the components If i ∈ [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 operations Six correspond closely to the six operations of $\mathcal{A}'''$ 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 operations are listed below As usual, we present them by listing a precondition and the effect on the state In addition, we define d(π), the doer of each step

In all cases, we assume that A ∈ act {U},

(a) **create**$_{i,A}$, origin(A) = i

(a1) Precondition
(a11) A ∉ i vertices_T
(a12) If parent(A) ≠ U, then parent(A) ∈ i vertices_T ∩ i committed_T
(a13) If (B A) ∈ seq and B ≠ A, then B ∈ i done_T

(a2) Effect
(a21) i vertices_T ← i vertices_T ∪ {A}
(a22) i status_T(A) ← 'active'

(a3) Doer i

(b) **commit**$_{i,A}$, A ∉ accesses, home(A) = i

(b1) Precondition
(b11) A ∈ i active_T
(b12) children(A) ∩ i vertices_T ⊆ i done_T

(b2) Effect
(b21) i status_T(A) ← 'committed'

(b3) Doer i

(c) **abort**$_{i,A}$, A ∉ accesses, home(A) = i

(c1) Precondition
(c11) A ∈ i active_T

(c2) Effect
(c21) i status_T(A) ← 'aborted'

(c3) Doer i

(d) **perform**$_{i,A,u}$, A ∈ accesses, x = object(A), u ∈ values(x),
home(A) = i, home(x) = i

(d1) Precondition
(d11) A ∈ i active$_T$
(d12) {B | V(x,B)} is defined} ⊆ proper anc(A)
(d13) u is the principal value of x in i V

(d2) Effect
(d21) i status$_T$(A) ← 'committed'
(d22) i V(x,A) ← 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 ∈ i committed$_T$

(e2) Effect
(e21) i V(x,parent(A)) ← i V(x,A)
(e22) i V(x,A) ← 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) ∩ i aborted$_T$ ≠ ∅

(f2) Effect
(f21) i V(x A) ← undefined

(f3) Doer i

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

(g1) Precondition
(g11) T' ≤ i T

(g2) Effect
(g21) M$_j$ ← M$_j$ ∪ T'

(g3) Doer i

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

(h1) Precondition
(h11) T' ≤ M$_i$

(h2) Effect
(h21) i T ← i T ∪ T'

(h3) Doer buffer

That is, any communication is allowed at any time, which sends any of the action summary information from i to j

Lemma 23 $\mathfrak{B}$ is an algebra, which is distributed over I using d

Proof Straightforward
□

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

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

vertices$_T$ ∩ {A origin(A) = i} ⊆ i vertices$_T$[b] ⊆ vertices$_T$

committed$_T$ ∩ {A home(A) = i} ⊆ i committed$_T$[b] ⊆ committed$_T$

aborted$_T$ ∩ {A home(A) = i} ⊆ i aborted$_T$[b] ⊆ aborted$_T$

i V[b] is the restriction of V to {(x,A) home(x) = i}

If i = 'buffer', then (T,V) ∈ h$_i$(b) exactly if (T,V) is computable in $\mathcal{A}'''$ and M$_j$[b] ≤ T for each j ∈ [k]

If (T,V) ∈ h$_i$(b), then we also say that (T,V) is i consistent with b

Lemma 24 For all i ∈ I, σ''' ∈ h$_i$(τ)
Proof Immediate from the definitions

□

Lemma 25 Assume i ∈ I Assume π' ∈ P, d(π) = i, π = h'''(π') ∈ Π''', a and a' are computable in $\mathcal{A}'''$ and $\mathfrak{B}$, respectively, a ∈ h$_i$(a') and a' ∈ domain(π') Then a ∈ domain(π)
Proof Let a be (T,V)

First, assume that π' is **create**$_{i,A}$, so that π is **create**$_A$ Then origin(A) = i Since a' ∈ domain(π'), A ∉ i vertices$_T$[a'] Since (T,V) is i consistent with a', A ∉ vertices$_T$, thus showing (a11) If parent(A) = U, then the fact that (T,V) is computable and Lemma 17 imply that parent(A) ∈ active$_T$, thus showing (a12) for this case On the other hand, if parent(A) ≠ U, then the precondition for π' shows that parent(A) ∈ i vertices$_T$[a'] i committed$_T$[a'] The fact that (T,V) is i consistent with a' implies that parent(A) ∈ vertices$_T$ committed$_T$ Thus, (a12) holds If (B,A) ∈ seq and B ≠ A, then the precondition for π' shows that B ∈ i done$_T$[a'] The fact that (T,V) is i consistent with a' implies that B ∈ done$_T$, thus showing (a13)

Second, consider π' = **commit**$_{i,A}$, so that π is **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 π' = **abort**$_{i,A}$, so that π is **abort**$_A$ This case is similar to the first half of the previous case

Fourth, assume π' = **perform**$_{i,A,u}$, so that π is **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 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) or (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 **release 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 **lose 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 26 Assume i, j $\in$ I Assume $\pi' \in$ P, d(π') = i, π = h'''(π') $\in$ 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(π') If b' = π'(a), then π(a) $\in h_j$(b')

Proof Let a = (T,V) and π(a) = (T',V') Lemma 25 implies that $a \in$ domain(π)

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 operations in turn

First, assume π' is of the form **create**$_{i,A}$, **commit**$_{i,A}$ or **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 **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 27 Assume i, j $\in$ I Assume $\pi' \in$ P, d(π') = i, h(π') = Λ, 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(π') If b' = π'(a'), then $a \in h_j$(b')

Proof Let a = (T,V)

First, assume that π' is **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 $M_i[a'] \leq T$ 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 $M_i[b'] \leq M_i[a'] \cup T'$ Therefore, each $M_i[b'] \leq T$, as needed

Next, assume that π is of the form $receive_{i,T''}$, 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 i consistent with a', $M_j[a'] \leq T$ Thus, $T' \leq T$ Therefore, $_j T[b'] \leq T$, as needed

□

Lemma 28 h''' and h_i, $i \in I$, form a local mapping from $\mathcal{B}$ to $\mathcal{A}'''$

Proof Immediate from Lemmas 24, 25, 26, and 27

□

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

Lemma 29 h''' is a simulation of $\mathcal{A}'''$ by $\mathcal{B}$

Proof Immediate by Lemma 28, Lemma 4 and Lemma 3

□

We are now ready to prove the main correctness theorem

Theorem 30 The mapping $h \circ h' \circ h'' \circ h'''$ is a simulation of $\mathcal{A}$ by $\mathcal{B}$

Proof Immediate from Lemma 29, Lemma 1 and Theorem 22

□

10 Acknowledgements

Many other people have contributed their ideas and efforts to this work Barbara Liskov suggested formal treatment of this area, and monitored proposed formalizations 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 Many of the ideas Gene Stark is developing for his thesis have found their way into the present paper Mike Fischer discussed some of the early attempts at formalization, and contributed several insightful suggestions Bill Weihl and Gene Stark contributed helpful criticisms of early drafts

References

[BG] Bernstein, P and Goodman, N
Concurrency Control Algorithms for Multiversion Database Systems
1982 ACM SIGACT SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, August 18 20, 1982, pp 209 215

[EGLT] Eswaren K P , Gray, J N , Lorie, R A and Traiger I L
The Notions of Consistency and Predicate Locks in a Database System,
CACM, Vol 19, No 11, November, 1976

[G] Goree, John
Internal Consistency of A Distributed Transaction System with Orphan Detection
M S Thesis, MIT Laboratory for Computer Sci , Cambridge, MA 1982 in progress

[KP] Kanellakis, P and Papadimitriou, C
On Concurrency Control by Multiple Versions
Proceedings of the ACM Symposium on Principles of Database Systems March 29 31, 1982, pp 76 82

[La] Lamport, L
Time, Clocks and the Ordering of Events in a Distributed System,
CACM, Vol 21, No 7, July, 1978

[LiS] Liskov, B and Scheifler, R
Guardians and Actions Linguistic Support for Robust, Distributed Programs,
1982 Ninth Annual ACM SIGACT SIGPLAN Symposium on PRINCIPLES OF PROGRAMMING LANGUAGES, Albuquerque, NM, January 25 27, 1982, pp 7 19

[M] Moss, J E B
Nested Transactions An Approach to Reliable Distributed Computing, Ph D Thesis,
Technical Report MIT/LCS/TR 260,
MIT Laboratory for Computer Science, Cambridge, MA 1981

[Ra] Randell, B
System Structures for Software Fault Tolerance
Proc Int Conf on Reliable Softw (April 1975), SIGPLAN Notices Vol 10 Nr 6, pp 437 457
Also in IEEE Trans Softw Eng Vol 1 Nr 2 (June 1975), pp 220 232

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

[S] Stark, E
Foundations of a Theory of Specification for Distributed Systems, Ph D Thesis, MIT Laboratory for Computer Science, Cambridge, MA 1982 in progress