A CONCURRENCY CONTROL THEORY FOR NESTED TRANSACTIONS*

Preliminary Report

C. Beeri†, P.A. Bernstein, N. Goodman
Harvard University

M.Y. Lai‡
Bell Laboratories

and

D.E. Shasha
Harvard University

0. INTRODUCTION

Concurrency control is the activity of synchronizing transactions that access shared data. A concurrency control algorithm is regarded as correct if it ensures that any interleaved execution of transactions is equivalent to a serial one. Such executions are called *serializable*. Serializability theory provides a method for modelling and analyzing the correctness of concurrency control algorithms [BSW, Pa].

The concept of nested transaction has recently received much attention [GR], [Mo]. In a nested transaction model, each transaction can invoke subtransactions, which can invoke sub-subtransactions, and so on. The natural modelling concept is the *tree log*. The leaves of a tree log are atomic operations executed by the underlying system. Internal nodes are operations (as seen by their parents) implemented as transactions (as seen by their children). Nodes are related by a partial order $<$, where $x < y$ means x executes before y [La].

*This work was supported by Rome Air Development Center, contract number F30602-81-C-0028, by the Office of Naval Research, contract number N00014-80-C-674, by the National Science Foundation grant number MCS79-07762, by Digital Equipment Corporation and by I.B.M.

†On leave from the Hebrew University.

‡Work performed while this author was at Harvard University.

We will use the following as a running example throughout the paper. Suppose user transactions issue the operations read a record, denoted r(rec), and write a record, denoted w(rec). r(rec) is implemented by fetching the disk page containing rec into a local buffer, denoted f(P), and then manipulating that local buffer. w(rec) is implemented by fetching the disk page containing rec into a buffer, updating the buffer, and then storing the page back to disk, denoted s(P).

Suppose that synchronization is by two-phase locking (2PL) [EGLT]. Each transaction locks a record before accessing it. To ensure that each write is atomic, a write operation locks the page containing its record before fetching it and releases the lock after storing it. A read does not lock the page.

Figure 1 shows a tree log of transactions that use these operations. The root is a monitor that runs transactions t_1 and t_2. The transactions access three records that are stored on the same page. Since the manipulation of a local buffer by a transaction does not interact with operations of other transactions, such manipulations are omitted from the log. In the log, $<$ is the transitive closure of the relationship denoted by the arrows.

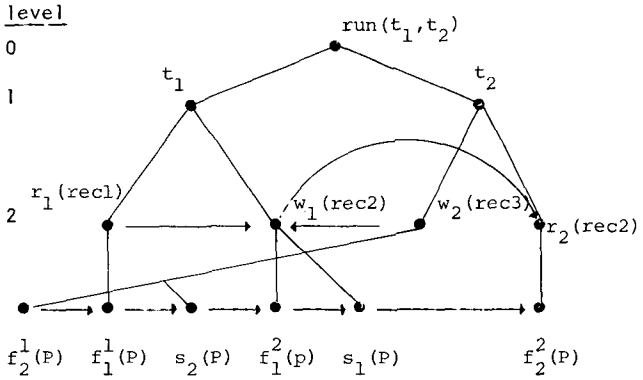

Figure 1. A tree log.

In standard serializability theory, this execution would be represented by a log consisting only of the leaves of this tree [Pa; BSW]. In this representation, the log would be judged as nonserializable: since $s_1(P) < f_2^2(P)$, t_1 must precede t_2; but since $s_2(P) < f_1^2(P)$, t_2 must precede t_1. No serial log containing t_1 and t_2 can satisfy both of these constraints.

Yet, by considering the rest of the tree, we see that the log really is serializable. It's equivalent to executing t_1 followed by t_2. The fact that $s_2(P) < f_1^2(P)$ is of no importance, because these operations are on different records. If we had executed $s_1(P) < s_2(P)$, the result would have been the same.

As shown by this example, the nested transaction concept is not limited to the case where application transactions are nested. In a multilevel system, user operations are expressed in a high level language; the system translates them, possibly in several stages, into executions of programs using low level operations. Every centralized database management system contains several levels. If the system is distributed, or a multiversion system, there is another level of translation--given an operation on a logical data item, it is necessary to choose a physical copy and to perform the appropriate operation on this copy. Synchronization in a multilevel system may take place at any one level, or at several levels (as in the example). The current state of the art [EGLT, BSW, Pa, RSL, SLR, BG81, BG82a] does not provide a framework for understanding concurrency control in such systems.

We mention two other areas where explicit or implicit nesting is a significant feature. The first area deals with concurrent operations on search structures such as B-trees and hash tables. There exists a wide variety of algorithms for performing concurrent searches and updates on such structures (see, e.g., [KW, LBY]). The nesting here is implicit, in that it is convenient to view the execution of the operations on several levels. There exists currently no general framework for understanding and proving the correctness of such algorithms. The second area is that of atomic abstract data types [LiWe, ScSp]. The idea here is to be able to specify and implement abstract data types that support concurrency for transactions that access them. Nesting is natural here, since an atomic data type can be viewed on at least two levels--the specification and the implementation.

In this paper we extend serializability theory to a nested transaction model. (A related proposal, using a different approach, appears in [Ly].) In addition to nesting, we extend the theory in two other ways. Firstly, we allow transactions to be arbitrary programs. Classical serializability dealt with transactions as straight-line programs, hence was unable to fully explain certain phenomena, e.g., the "phanton problem" [EGLT, BGL]. Secondly, we allow arbitrary operations, and not just reads and writes. This generalization is obviously needed for the treatment of arbitrary data types, but is also very useful in all multilevel systems. Our extension enables us to prove the correctness of a wide variety of concurrency control algorithms. An important feature of the extension is that standard serializability theoretic proofs can be applied without modification in nested transaction proofs.

In order to make our results widely applicable, it was necessary to derive them in the context of an abstract model of computation, where only the details that are relevant to concurrency control theory are made explicit. We view this abstract model as an important contribution of this paper. A general description of the model is presented in Section 1. Sections 3-5 then present in details the main components--the storage system, the transactions and the computations. In Section 6 we study substitution as a tool for proving equivalence of computations. In Section 7 we study conflict preserving serializable logs and in the next section we deal with various choices of conflict predicates. The results in these two sections are of primary significance since all practical concurrency control algorithms are conflict driven. Section 9 presents examples, and in Section 10 we present conclusions and directions for future research.

1. THE MODEL

As stated previously, the high level abstraction of the model is a major factor in making the results widely applicable. In this section we present a general description of our model, and we explain some of our decisions regarding various features.

A *computing environment* consists of two components, a *system* component and *transaction management* component. The system component represents the storage system; it contains all shared data and the processes that manage it and support operations to access the data. The transaction management component contains the transactions and facilities for invoking subtransactions and for scheduling.

We represent the data by a system state. More specific concepts, like variables and their values, location and distribution details, are not used. These are needed only for the description of specific applications of the theory. Similarly, dynamic behavior of the system and the transactions is described by sets of computations, rather than by programs. These two abstractions contribute significantly to the overall simplicity of our model.

The two components exchange information as follows. Requests to execute operations are passed from the transactions to the system. For each completed operation, a return value is sent by the system to the issuing transaction. The details of how information is passed are irrelevant to the model.

In a distributed system, not every pair of events is necessarily ordered in time [La]. We assume that the events of an execution are related by a partial order, called the *execution order*. In any real computing environment, this partial order has the properties: events observed by the same

process are ordered and a message is sent before it is received.

Two types of events are of interest to us: the initiations and the completions of operations. In principle, the execution order may relate any two events to each other. However, we use only the partial order defined on the operations, where operation o_1 *precedes* operation o_2 in the execution order, or o_1 *happens before* o_2, if o_1 completed before o_2 was initiated. If neither of the two operations proceeds the other, then they are *concurrent*.

This assumption implies that a transaction's behavior at any time depends only on the operations that completed and their return values. Obviously, there is no significance to the relative order of operation initiations that occur in a time interval with no completions. There are cases where transactions seem to depend on completion order in their choice of one of several alternatives. However, in all these cases that we are aware of, the choice is related to issues such as efficiency, termination, fairness, etc. From the point of view of concurrent correctness, all the alternatives are correct for each order of completions that is being considered. Hence, by introducing nondeterministic behavior, our assumption holds.

As further justification of our assumption, we note that in serializability theory, system computations are always assumed to be (equivalent to) serial, the goal is to serialize the execution of transactions and the mechanism employed delays the initiation of some operations until other operations complete. These are all expressed in terms of "happened before". Our success in applying our results to many algorithms witnesses the validity of the assumption.

This assumption implies that not all the timing information that may be available is used. It follows that an execution order may correspond to several actual executions, possibly with different results. This introduces *indeterminacy* into the system and transaction description. As noted above, it may even be convenient to assume nondeterministic behavior. We will usually use deterministic terminology, for convenience, but none of our results requires deterministic behavior.

We now describe the basic building blocks that are used in the following sections. Let $S=\{s_1,s_2,\ldots\}$ be a set of *sytem states*; OP be a set of operations; V be a set of *return values*; R be the set of return value functions and POS be the set of partial orders on OP. The remaining components are the *system* S, the *transactions* $T=\{t_1,t_2,\ldots\}$, and the set of *hierarchical computations* HC. These will be defined later.

Operations are issued by transactions and executed by the system. We assume that each operation is associated with a unique transaction and that occurrences of the operation in an execution of the transaction can be distinguished. In practice, a system is specified using a set of generic operations. The set OP of operations is obtained by attaching indices that identify the transaction and the occurrence in the execution.

We are interested only in subsets of OP that occur in finite computations. Let 2^{OP}_{fin} denote the collection of all finite subsets of OP.

It is a significant feature of our approach that a transaction is also regarded as an operation. Thus, our set of operations contains both low level and high level operations, though we do not make this distinction formally. We assume that there exists a one-to-one correspondence between the set T and a subset OP_T of OP.

POS is the set of partial orders on finite subsets of OP. We use the abbreviation *poset* for partially ordered sets. For convenience, we usually do not write explicitly the domain of a partial order, and it is to be inferred from the context. The notation $<[X]$ is used when we want to make the domain X explicit, e.g., when we want to restrict $<$ to the domain X.

V is the set of values that can be returned by the system for completed operations or by a transaction when it terminates. A value may be anything between a success message and a complex report printed by a transaction. The *null* value $\perp$, is a legal return value.

For technical reasons, we want every poset of members of $V-\{\perp\}$ to be in V. Let $V_0 \subseteq V$, $\perp \in V_0$, be a set of *simple* values, and let $V = V_0 \cup \{\text{all finite posets on } V_0 - \{\perp\}\}$. Every finite poset of $V-\{\perp\}$ can then be identified with a member of V. The poset $(Q,<) = (\{q_i,<_i\},<)$ is identified with $(\bar{Q},\bar{<}) \in V$, where

$$\bar{Q} = \cup q_i \ ,$$

$$\bar{<} = (\cup_i <_i) \cup \{(v_i,v_j) \mid v_i \in q_i, v_j \in q_j, (q_i,<_i) < (q_j,<_j)\} \ .$$

Each $r \in R$ is a function that associates return values with operations in a finite set. As for partial orders, we usually do not write its domain, and we use the notation $r[X]$ to make the domain X explicit.

2. THE SYSTEM

2.1 System Computations

We describe the system by the set of computations it can execute. In general, operations may arrive concurrently and may be processed concurrently. Thus, a computation involves a poset of operations. Also, in a distributed system, the concept of global state is not well defined unless the system is quiescent [ABG, FGL]. For these reasons, in general, it does not suffice to describe the effect of each individual operation on the system state; a description of concurrent operations must be given.

Formally, a *system* $\mathcal{S}$ is a subset of $S \times 2^{OP}_{fin} \times POS \times R \times S$, that satisfies the conditions (C1) and (C2) below. Tuples in this product will be called *system tuples*; the members of $\mathcal{S}$ will be called *system computations* or $\mathcal{S}$*-computations*.

A tuple $c_{\mathcal{S}} = (s_1, O, <, r, s_2) \in \mathcal{S}$ states that starting from state s_1, $\mathcal{S}$ can execute the operations in O such that the execution order is $<$, the return values are specified by r and the final state is s_2. For $o_1, o_2 \in O$, we say that o_1 *happened before* o_2 or that o_1 *precedes* o_2 in $c_{\mathcal{S}}$ if $o_1 < o_2$. If $o_1 \not< o_2$ and $o_2 \not< o_1$ then they are *concurrent* in $c_{\mathcal{S}}$.

Before we can state the conditions on $\mathcal{S}$, we need to introduce some concepts. An $\mathcal{S}$-computation $(s_1, O, <, r, s_2)$ is *serial* if $<$ is a total order on O. It is *atomic*, if the partial order $<$ can be extended to a total order, i.e., for some total $<'$, where $< \subseteq <'$, $(s_1, O, <', r, s_2) \in \mathcal{S}$.

In this paper, we will principally use atomic computations. The computations that involve low level operations, i.e., those executed by the underlying system will be assumed to be atomic. In an application of our theory, these will be specified in the form described here, as posets. Higher level operations, i.e., those corresponding to transactions, will be specified individually. If nonserial executions involving these operations will be included in the system description, they will have to be proven to be atomic as part of the correctness proof.

In our running example, the sequence of page operations with their associated return values becomes an $\mathcal{S}$-computation when initial and final states are given. Each level 2 operation has a well-defined semantics, defined by its effect when it is executed alone. Hence serial computations involving reads and writes on records are well defined. The poset given in level 2 of our tree, i.e., $r_1, w_2 < w_1 < r_2$, in which r_1 and w_2 are concurrent, if augmented with states, may be considered a system computation. It is part of the task of proving the correctness of the tree to show that it is atomic.

In our approach, atomicity is not a basic concept. Rather, it is definable in terms of serial computations and equivalence to such computations. For example, a database system may regard the operating system interface as atomic. In practice, these are complex operations and in a multiprocessing environment they may be interleaved. All the operating system guarantees is that they can be considered as if executed serially.

Let us compare atomicity as defined here to the standard assumptions found in the literature. For centralized systems, system computations are assumed to be serial [BSW, Pa]. For distributed systems, this is generalized to assume a partial order $<$, where every pair of conflicting operations is $<$-related. Now, suppose that operations that are not $<$-related are allowed to be arbitrarily interleaved. Then this assumption is not a sufficient basis for the theory, because an interleaved execution of commuting operations is not necessarily equivalent to a serial one. Thus, an explicit assumption about atomicity must be made. Once such an assumption is made, there is no need to assume that conflicting operations are $<$-related. If it is needed, it can be assumed to be implied by atomicity, since we can work with some equivalent serial computation.

We can now state the conditions $\mathcal{S}$ is required to satisfy.

Let $O_1, O_2 \subseteq OP$, such that $O_1 \cap O_2 = \emptyset$, let $<_1$ and $<_2$ be partial orders on O_1, O_2, resp. and let $<_1 \circ <_2 = <_1 \cup <_2 \cup O_1 \times O_2$ (i.e., $<_3$ agrees with $<_1$ on O_1, with $<_2$ on O_2, and requires all operations of O_1 to precede those of O_2). Let $r \in R$.

(C1) (Composition axiom) If, for some s_1, s_2, s_3, $c^1_{\mathcal{S}} = (s_1, O_1, <_1, r, s_2)$ and $c^2_{\mathcal{S}} = (s_2, O_2, <_2, r, s_3)$ are in $\mathcal{S}$, then so is their *composition* $c_{\mathcal{S}} = c^1_{\mathcal{S}} \circ c^2_{\mathcal{S}} = (s_1, O_1 \cup O_2, <_1 \circ <_2, r, s_3)$.

(C2) (Decomposition axiom) If, for some s_1, s_3, $c_{\mathcal{S}} = (s_1, O_1 \cup O_2, <_1 \circ <_2, r, s_3) \in \mathcal{S}$, then there exists an $s_2 \in \mathcal{S}$ such that $c^1_{\mathcal{S}} = (s_1, O_1, <_1, r, s_2)$ and $c^2_{\mathcal{S}} = (s_2, O_2, <_2, r, s_3)$ are in $\mathcal{S}$. Further if $c_{\mathcal{S}}$ is atomic, so are $c^1_{\mathcal{S}}$ and $c^2_{\mathcal{S}}$.

Condition (C2) states that if an $\mathcal{S}$-computation contains a point such that each operation either precedes or follows this point, then the computation is a composition of two computations that have a common state at this point. We will refer to this state s_2 as an *intermediate state*. We do not assume that decomposition is unique. However, for convenience, we will use this axiom as if there is a unique decomposition, hence a unique intermediate state is associated with the breakpoint. Our results do not depend on uniqueness.

2.2 Commutativity

Let o_1, o_2 be operations, s a state. We say that o_1 and o_2 *commute w.r.t.* s if the order of o_1 and o_2 in any serial $\mathcal{S}$-computation from s can be reversed. That is, for all r, s', $(s, \{o_1, o_2\}, (o_1, o_2), r, s') \in \mathcal{S}$ iff $(s, \{o_1, o_2\}, (o_2, o_1), r, s') \in \mathcal{S}$. The two operations (generally) *commute* if they commute w.r.t. all states. If o_1 and o_2 do not commute, then they *conflict*.

Concurrency control algorithms are usually designed with general commutativity in mind. State based commutativity potentially offers more concurrency, but is more difficult to use. It also incurs more overhead, since to decide if two operations commute or conflict, the scheduler needs to know the intermediate state preceding them. If, however, operations are known to generally commute, their order can be reversed whenever they are adjacent.

A *general conflict predicate* is a set of pairs of operations such that it contains all pairs that conflict, and possibly other pairs as well. Conflict predicates are typically used by schedulers when they need to decide if operations should be allowed to execute concurrently or not. The property of conflict predicates that we need is stated after the next lemma.

LEMMA. *Let* $<$ *be an acyclic binary relation on a set* X *and let* $<_1$ *and* $<_2$ *be two total orders on* X *that contain* $<$. *Then* $<_2$ *can be obtained from* $<_1$ *by transpositions of adjacent, non* $<$*-related, elements.* □

PROPOSITION 1. *Let* CON *be a general conflict predicate, and let* $(s_1,O,<,r,s_2)$ *be a serial* $\mathcal{S}$*-computation. Then, for every total order* $<'$ *that contains* $< \cap$ CON, $(s_1,O,<',r,s_2)$ *is a serial* $\mathcal{S}$*-computation.*

Proof. We show that if operations x and y, where $(x,y) \notin$ CON, are adjacent in a serial $\mathcal{S}$-computation, then their order can be reversed. The claim then follows from the definition of a conflict predicate and the lemma. To show that, we use the decomposition axiom to isolate x and y, their commutativity to reverse their order, then the composition axiom. □

Can we extend the concept of commutativity so as to obtain more concurrency, retaining the advantages of low overhead for the scheduler and the property expressed in Proposition 1? It turns out that we can. The idea is to use information that is made available during the computation.

For example, several recent papers [LiWe, ScSp] suggest the use of operation-return value commutativity. For operations o_1 and o_2, and return values v_1 and v_2, we say that (o_1,v_1) and (o_2,v_2) *commute* if the order of o_1 and o_2 can be reversed in every serial $\mathcal{S}$-computation in which their return values are v_1 and v_2, resp. Obviously if o_1 and o_2 commute, then for every v_1 and v_2 (o_1,v_1) and (o_2,v_2) commute. The converse does not hold. Thus, operation-return value commutativity offers more concurrency than general commutativity, yet it retains the advantages of the latter.

Let $f(o, s_1,s_2,v)$ be a function whose argument types are an operation, two states and a return value. We can associate with each occurrence of an operation o in a serial $\mathcal{S}$-computation the value $f(o,s_1,s_2,v)$, where s_1 and s_2 are the intermediate states that precede and follow o, and v is o's return value. We say that f is a *commutativity parameter* if whenever two adjacent operations commute in a computation, their associated f-values remain the same when their order is reversed.

Given two f-values f_1 and f_2, we say that (o_1,f_1) and (o_2,f_2) *commute* if for every serial computation on o_1, o_2, if the corresponding f-values are f_1 and f_2, resp., then o_1 and o_2 commute w.r.t. the state preceding them. We will refer to this type of commutativity as operation-f commutativity. It can easily be seen that if o_1 and o_2 commute then (o_1,f_1), (o_2,f_2) commute, but the converse does not hold. Thus, more concurrency can be obtained by using operation-f commutativity. The definition of a conflict predicate and Proposition 1 can also be extended easily. In this paper, the results are stated in terms of general commutativity and conflict predicates, but they apply equally (with the appropriate changes) to operation f commutativity and the corresponding conflict predicates.

The function $f(o,s_1,s_2,v) = v$ is a commutativity parameter. In many systems, the write set of an operation, i.e., the set of data item affected by it is also a valid parameter. For low level operations, the write set is usually part of the operation specification, so operation-f commutativity reduces to commutativity. However, for high level operations, i.e., for transactions, the write set is determined only during operation, so the use of the write set as a commutativity parameter increases the level of concurrency. Finally, we note that a vector of commutativity parameters is also a commutativity parameter. The semi-queue example presented later uses the return value and write set as a parameter. This example illustrates how such parameters can be effectively used for scheduling.

3. TRANSACTIONS AND SUBTRANSACTIONS

A transaction is any (distributed) program with any number of agents executing concurrently on its behalf. As for the system, we choose the most general representation, namely a set of computations. A transaction has only one meaningful initial state. It may have more than one final state, but we assume that its return value contains all the relevant information about its termination status. Hence transaction computations do not contain states.

Formally, a transaction t is a subset of $2^{OP}_{fin} \times POS \times R \times V$, that satisfies the condition (C3) below. A tuple $c = (O,<,r,v)$ is called a *transaction tuple*. If $c \in t$, it is called a t-*computation*; usually we use c_t to denote t-computations. A tuple as above represents a computation where O is the operation set, $o_1 < o_2$ means that t has received the return value for o_1 before it initiated o_2, r is the return value function and v is the value returned by t itself. Note that $c_t \in t$ does not imply that if t sends the operations of O to $\mathcal{S}$, $\mathcal{S}$ will respond with the return values specified by r.

The condition on t is

(C3) If $(O,<,r,v) \in t$ and $< \subseteq <'$, then $(O,<',r,v) \in t$.

As an example, assume that o_1 and o_2 are initiated concurrently and that t's control structure requires o_3 to be initiated if o_1 returns the value 1. If o_1 returns first, with value 1, the resulting partial order will be $o_1 < o_3$. If o_2 returns first, then o_1 returns with value 1, then o_3 will still be initiated, and the partial order will be $o_1 < o_3$, $o_2 < o_3$. Thus the condition essentially states that a transaction's behavior be, in a sense, independent of the relative speeds or order of processing of concurrent operations by the system. An initiation of an operation depends on some previously executed operations and their results, and is not invalidated if some other concurrent operations have also terminated.

For a given (O,r,v), there may exist several partial orders that complete it to a t-computation. A partial order $<$ is called a *transaction order*

for t and (O,r,v) if (O,<,r,v) $\in$ t but, for every <' $\subsetneq$ <, (O,<',r,v) $\notin$ t. Intuitively, a transaction order represents the dependencies in a computation implied by the control structure of t. For each operation o, it specifies those operations and return values that caused t to initiate o. In most applications, we expect the transaction order to depend on (O,r) only.

A transaction order for t will be denoted by $<_t$. We make no assumption about uniqueness of a transaction order for t and (O,r,v) or about t being deterministic; our results do not depend on such assumptions. For convenience, we will usually use deterministic notation, e.g., we will refer to *the* transaction order.

To model the fact that transactions can invoke subtransactions, we assume the existence of a *translation* that associates operations and transactions. The translation is a one-to-one function between the set T of transactions and a subset OP_T of OP. Thus, each transaction can be considered as an implementation of an operation. We denote by t(o) the transaction associated with operation o.

Assume operation o is initiated by a transaction t. If it is not in OP_T, it must be sent directly to $\mathcal{S}$ for execution. If it is in OP_T, then the subtransaction t(o) may be invoked instead. Its return value is considered by t as the return value of o. From t's viewpoint, the way o is executed is irrelevant; it need not be aware of the existence of subtransactions.

There are two aspects to correctness: sequential correctness which applies when a transaction executes alone, and concurrent correctness which applies when executions of several transactions are interleaved. To isolate the issues that are relevant to concurrent correctness, we assume in this paper sequential correctness of the translation.

Sequential correctness is not necessarily easy to prove. If the meaning of an operation o is defined by the effect of t(o) on the system, there is nothing to prove. However, if an abstract specification is given for o, i.e., the specification of $\mathcal{S}$ describes the effect of applying o on some states, and t(o) is offered as an implementation, then its correctness must be proved. Proof methods for sequential correctness are well known [MaPn] and will not be discussed here.

Formally, sequential correctness is expressed as follows:

(C4) (Sequential correctness axiom) Let o be an operation, t(o) its translation. For all s_1, s_2,O,<,r,v, if $(s_1,O,<,r,s_2)$ is an atomic $\mathcal{S}$-computation and (O,<,r,v) is a t(o)-computation, then (s_1,o,ϕ,v,s_2) is also an $\mathcal{S}$-computation (obviously atomic).

Note that we assume that whatever t(o) does can also be achieved by applying o directly. The converse, i.e., that whatever o does can be achieved by applying t(o) is not assumed. As remarked in [LiWe], a correct implementation of an operation on an abstract data type is not required to generate all possible executions of the operation. Similarly, it is well known that practical concurrency control algorithm only generate subsets of the serializable executions.

We emphasize that the operations of t(o) are assumed to be executed atomically. If they are allowed to invoke subtransactions, whose executions may be arbitrarily interleaved, sequential correctness does not apply.

In our running example, the operation w_1(rec2) is translated into f_1^2(P) followed by s_1(P). The transaction order is $f_1^2 < s_1$. If records do not move from page to page, this a straight-line transaction, it never changes. Note that (C4) can be applied to r_1, w_1 and r_2, but not to w_2.

4. COMPUTATION FORESTS

Assume several transactions are executing. Since each may invoke subtransactions, the execution generates a forest, where each node is labeled with an operation and, for each internal node, the operation is associated with a transaction, i.e., it is in OP_T. Denote the set of all finite forests so labeled by $\mathcal{F}$. For convenience, we identify each node with its label.

The set HC is a subset of $S \times \mathcal{F} \times POS \times R \times S$, satisfying the conditions stated below. A tuple $c = (s_1,F,<,r,s_2)$ is called a *hierarchical tuple*. Note that if F is simply a set of nodes, it is also a system tupe. If it is in HC, it is called a *computation forest*. In c, s_1 and s_2 are the initial and final states, resp., F is the forest of operations, < is the execution order, and r is the return value function. Note that r associates a return value with each node, including the roots. By definition of $\mathcal{F}$, each internal node o has a translation t(o).

Given a computation forest c, we can add a root whose task is to run the transactions at the roots of F. The value it returns is the poset of return values of the roots of F. The forest F then becomes a tree, which we denote by T, and c becomes a *computation tree*, c'. The computation forest c and the computation tree c' are equivalent (according to the definition given below). The tree viewpoint is especially convenient when the roots of F must satisfy some externally specified timing constraints. These constraints can be considered as the transaction order of the tree root. In this paper, the discussion will usually be presented in terms of tree. Everything applies to forests as well.

We introduce some notation. For a computation tree c, the set of leaves is leaves(T) or leaves(c); the root is root(T) or root(c). For a node x, the subtree rooted at x is T_x, and its leaves are leaves(x). Similarly, if X is a set of nodes, the forest rooted at X is T_X and its leaves are leaves(X). The set of descendents of x is desc(x) and the set of childen (i.e., immediate descendents) is child(x). The lowest common ancestor of nodes x and y is lca(x,y). Nodes x and y are *incomparable* if lca(x,y) is

neither x nor y. A set of pairwise incomparable nodes is a *partial front*; a *front* is a maximal partial front.

For a front M, c *restricted to* M, denoted c[M], is the tuple $(s_1, M, <[M], r[M], s_2)$. Note that c[M] can be viewed as a system tuple, but it is not necessarily an $\mathcal{S}$-computation. It can also be viewed as a "flat" hierarchical tuple, but it is not necessarily a computation forest. For an internal node x, c *restricted to* x, denoted c[x], is the transaction tuple (child(x),<[child(x)], r[child(x)],r(x)). It is not necessarily a t(x)-computation. We denote c[leaves(c)] by $c[\mathcal{S}]$ and call it the *system projection* of c. It is the part of c that involves the system.

The conditions on HC follow.

(C5) (Downward order-tree compatibility) Let x,y ∈ T, and p ∈ desc(x), q ∈ desc(y). If x < y then p < q.

(C6) (Transaction validity) For each internal node x, c[x] is a t(x)-computation.

(C7) (System validity) The system projection of c, $c[\mathcal{S}]$, is an atomic $\mathcal{S}$-computation.

The meaning of (C5) is obvious. If t(x) completes before t(y) begins, then each operation of t(x) completes before any operation of t(y) begins. Condition (C6) requires each internal node to see a valid t(o)-computation. Condition (C7) requires the system component to be a valid atomic $\mathcal{S}$-computation. The atomicity requirement reflects our goal of dealing with serializable computations. If atomicity is not guaranteed for the base level computations, nothing much can be done to guarantee serializability for transactions.

PROPOSITION 2. *Let c be a computation tree. The partial order < of c can be extended to a partial order <' such that the result c' is a computation tree and for all nodes x and y, if* p <' q *for all* p ∈ child(x), q ∈ child(y) *then* x <' y.

Sketch of Proof. The proof relies on the fact that (C3) allows us to extend a partial order on a tree without violating (C6). □

Intuitively, the proposition states that if all operations in the implementation of x precede those in the implementation of y, then the order can be "stretched" so that x precedes y. Assumption (C3) is crucial for this to hold. In the sequel, we will assume that < satisfies both downward compatibility (by (C5)) and upward compatibility (by the proposition). Under this condition, the order on the leaves determines the order on T.

5. EQUIVALENCE AND CORRECTNESS

Two computations forests c_1 and c_2 are *equivalent*, denoted $c_1 \equiv c_2$, if they have the same initial and final states and the same poset of (non-null) return values from the roots. Note that forests with different sets of roots may be equivalent, since some roots may return a null value and some may return posets of values. Any forest is equivalent to the tree obtained from it by adding a dummy root, as explained previously.

We can now deal with various notions of correctness. Since the set of correct computations should be closed under equivalence, correctness can be specified as follows: Let CE be a set of computations. A computation c is CE-*correct* if $c \equiv c'$ for some $c' \in CE$.

Serializability is a special case. We say that a computation tree c is *serial* if < is a total order on the children of each internal node. Let SERIAL be the set of serial computation. A computation is *serializable* if it is SERIAL-correct. The set of serializable computation is denoted by SR.

Example 1. Let us consider how the simple model of (flat) serializability described in [BSW,Pa] fits into our framework. A system state is an assignment of values to a fixed set of variables. Transactions are striaght line programs with no subtransactions. The low level operations are read(x) and write(x). The value to be written by write(x) is not known. Since an operation is supposed to effect a known transformation on the state, it is assumed that a write(x) executed by transaction t_i, writes a value determined by a function f_i whose arguments are the previous values read by t_i, such that f_i is always different from f_j and f_i delivers different values for different arguments. There is an implicit assumption that each read and write is atomic. It is also assumed that a read(x) and write(x) are always related by the partial order, and for each read(x), there is a unique last write(x) that precedes it (unless the read actually accesses the original value of x).

Transaction t_i reads x from transaction t_j if t_j performs the last write on x before t_i reads x. The set of *live* transactions of a computation c is the smaliest set that contains

(i) The transactions of c that return a non-null value, and
(ii) The transactions that write the final value of some x,

and such that if t_i is in the set and t_i reads x from t_j than t_j is in the set.

Under these assumptions, we have the following.

THEOREM 1 [BSW,Pa]. *Two computations are equivalent iff they have the same set of live transactions and the same reads from relation on the live transactions.* □

The if direction can be generalized to our model. The only if direction does not necessarily hold, since transactions are not required to be straight line programs, and the semantics of operations may be more detailed. □

In our running example, the given tree is equivalent to a serial computation tree in which

t_1 precedes t_2 (for any assignment of values to the records). Usually, not all computation trees are in SR. To ensure serializability, a concurrency control mechanism is used to restrict the set of trees that are allowed to occur. A proof of correctness for such a mechanism show that this restricted set is contained in SR. The rest of the paper is devoted to proof techniques and their application. We conclude this section with a discussion of a property of trees that is very useful for proving equivalence.

Let $c = (s_1, T, <, r, s_2)$. For a front M, T_M is the subforest rooted at M. Let $c_M = (s_1, T_M, <[T_M], r[T_M], s_2)$. We call c_M the *subcomputation* rooted at M. Note that $c_M[\mathcal{S}] = c[\mathcal{S}]$ and that c_M satisfies (C5)-(C7), i.e., it is a computation forest.

Let T/M denote the tree T with the proper descendents of M removed, and let $c/M = (s_1, T/M, <[T/M], r[T/M], s_2)$. We call c/M the *remainder computation* (modulus M). Note, however, that c/M may fail to satisfy (C7), hence it is not necessarily a computation tree.

THEOREM 2. *Let* c *be a computation tree and let* M *be a front. Then* c[M] *is an atomic* $\mathcal{S}$*-computation iff* c/M *is a computation tree and* $c \equiv c/M$.

Proof. It is easy to see that c/M satisfies conditions (C5) and (C6). Since $c/M[\mathcal{S}] = c[M]$, c/M satisfies condition (C7), and hence is a computation tree, iff c[M] is an atomic $\mathcal{S}$-computation. If c/M is a computation tree, then obviously $c \equiv c/M$. □

The typical way the theorem is applied is by showing that c_M, the subcomputation rooted at M, can be reduced to its roots, i.e., that $c_M \equiv c[M]$ (where c[M] is viewed as a hierarchical tuple). Indeed $c_M \equiv c[M]$ holds iff c[M] is an atomic $\mathcal{S}$-computation (when viewed as a system tuple). We thus obtain the following version of the theorem.

THEOREM 2'. *Let* c *be a computation and let* M *be a front. Then* c/M *is a computation forest, equivalent to* c, *iff* $c_M \equiv c[M]$. □

COROLLARY 1. *Every serial computation* c *is equivalent to its root, i.e.,* $c \equiv c[\text{root}(c)]$.

Sketch of Proof. The proof uses (C4) to reduce the tree, going from the leaves toward the root. □

By the corollary, every serial computation is equivalent to a single node computation. To prove that $c \in SR$, it suffices to reduce it to its root. The techniques we present rely on this observation. Note that Theorem 2, in either form, cannot be considered as a tool for proving equivalence or serializability. Rather, it is a framework for applying such tools. Its significance and power lies in its generality, since it allows any method for proving equivalence to be used, and it also allows different methods to be used in reducing layers of the tree.

The complexity of determining CE-correctness depend, of course, on the details of the system and the transactions. It is known to be NP-complete for the flat read/write model [Pa]. The complexity for the general case is unknown.

6. EQUIVALENCE BY SUBSTITUTION

Theorem 2 can be viewed as stating that the substitution of a subcomputation rooted at a front by another preserves equivalence. In this section we present a general theory for substitution.

6.1 Partial Orders on Trees

We will need a few definitions and technical results about partial orders on trees. These are collected here for convenience.

Let $<$ be a partial order. For sets x,y, we write $x < y$ if for all $x \in X$, $y \in Y$, $x < y$ holds. We write $X \tilde{<} Y$ if for some $X \in X$, $y \in Y$, $x < y$ holds.

Let $<$ be a partial order on leaves(T). We define two binary relations on T, derived from $<$, the *pull up* of $<$, denoted $<\uparrow$, and the *strong pull up* of $<$, denoted $\tilde{<}\uparrow$. For incomparable nodes x and y, $x <\uparrow y$ if leaves(x) $<$ leaves(y); $x \tilde{<}\uparrow y$ if leaves(x) $\tilde{<}$ leaves(y).

LEMMA 2. *For* $<$ *as above,* $<\uparrow$ *is a partial order on* T. □

In our running example, let $<$ be the given partial order restricted to the leaves. Then leaves(w_2) < leaves(w_1); leaves(t_1) $\tilde{<}$ leaves(t_2) but *not* leaves(t_1) < leaves(t_2). Also, $r_1 <\uparrow w_1 <\uparrow r_2$ and $t_1 \tilde{<}\uparrow t_2 \tilde{<}\uparrow t_1$. There is no $<\updownarrow$ relationship between t_1 and t_2.

Let $<$ be a partial order on T. The *pull down* of $<$, denoted $<\downarrow$, is defined by: $x <\downarrow y$ if for some ancestors p of x and q of y, $p < q$. In general, $<\downarrow$ need not be acyclic.

Let c be a computation. For each internal node o, there exists a t(o)-transaction order on child(o), denoted by $<_{t(o)}$. The order

$$<_t \; = \; (\bigcup_{o \in T} <_{t(o)})\downarrow$$

is called the *transaction order* for c. (Actually more than one can exist; however, as usual, we use deterministic notation.) The reference to $<_t$ as an order is justified by the following lemma.

LEMMA 3. *The binary relation* $<_t$ *defined above is a partial order, contained in* $<$. □

LEMMA 4. *For* $x,y \in \text{child}(o)$, *the following are equivalent*

(1) $x <_{t(o)} y$,

(2) $x <_t y$,

(3) $x (<_t\downarrow)\uparrow y$,

(4) $x (\tilde{<}_t\downarrow)\uparrow y$. □

By the lemma $<_t$ is closed w.r.t. pull down and both types of pull up. To determine if $x <_t y$, all that is needed is to observe the transaction order between the children of $lca(x,y)$ from which x and y are descended.

In the running example, assume that the transaction order for each write requires the fetch to precede the store and that $r_1 <_{t_1} w_1$ and $w_2 <_{t_2} r_2$, but there is no predefined order on t_1, t_2. Then $<_t$ is given by $f_1^1 <_t f_1^2 <_t s_1$; $f_2^1 <_t s_2 <_t f_2^2$.

Let $<$ be a partial order on $leaves(T)$, and let x be a node. We say that $<$ *separates* x or that x *is separated* by $<$, if for each $y \in leaves(T) - leaves(x)$, either $y < leaves(x)$ or $leaves(x) < y$. A set of nodes is separated by $<$ iff each node in the set is separated by it. We will deal with separation of sets only for partial fronts.

Separability of a node means that the leaves under it are executed without interleaving with other leaves. Thus as far as an outside observer can tell, the transaction at the node is executed atomically. (It may still be the case, of course, that operations of subtransactions are interleaved.)

LEMMA 5. *If* $<$ *separates* M *(where* M *is a partial front) than for all* $x,y \in M$, *either* $leaves(x) < leaves(y)$ *or* $leaves(y) < leaves(x)$. □

COROLLARY 2. $(\tilde{<}\uparrow)[M] = (<\uparrow)[M]$, *and it is a total order on* M. □

6.2 Substitutions

Our interest in the property of separability is due to the fact that it is a sufficient condition for substitution to preserve equivalence. In the following, let $c = (s_1,T,<,r,s_2)$ be a fixed computation tree and let x be a node that is separable by $<$.

LEMMA 6. *For* c *and* x *as above,* $c[S] = c_S^1 \circ c_S^2 \circ c_S^3$ *where* c_S^i *is an atomic* S*-computation* $i=1,\ldots,3$, *and* $c_S^2 = c_x[S]$. □

In the following, we denote the initial state of $c_x[S]$ by s_1' and its final state by s_2'.

LEMMA 7. *For* c *and* x *as above,* $c_x = (s_1',T_x', <[T_x], r[T_x], s_2')$ *is a tree computation.* □

We recall that by Proposition 2, we may w.l.o.g. assume that for every node y, if $y < leaves(x)$ than $y < x$ and if $leaves(x) < y$ than $x < y$.

Let $d_x = (s_1',T_x',<_x,r_x,s_2')$ be a computation tree, where s_1', s_2' are the initial and final states of c_x mentioned above, T_x' is a tree with root x, and $r_x(x) = r(x)$. The *substitution of* d_x *for* c_x *in* c, denoted $c[c_x \Leftarrow d_x]$, is

$$c' = (s_1,T',<',r',s_2) ,$$

where

(i) T' is T with T_x replaced by T_x'.

(ii) $<'$ is equal to $<$ on $T - T_x$, and is equal to $<_x$ on T_x'. For $y \in T - T_x$, if $y < x$ $(x < y)$ then $y <' T_x'$ $(T_x' <' y)$.

(iii) r' is equal to r_x on T_x' and is equal to r elsewhere.

THEOREM 3 (Substitution Theorem) *Let* c *be a computation tree,* x *a node separable by* $<$, *and let* d_x *be a computation tree with root* x *such that* $d_x \equiv c_x$. *Then* $c' = c[c_x \Leftarrow d_x]$ *is a computation tree and* $c \equiv c'$. □

Note that, in general, equivalent computations are not required to have the same root. To perform substitution, we also need equality of roots. The reason is that if the parent transaction sent the operation x in the old computation, it sends x in the new computation as well.

Let c, x, c_x and d_x be as above, and let $c' = c[c_x \Leftarrow d_x]$. In c', x is separable, so we can substitute c_x for the subcomputation d_x. It is easy to see that $c = c'[d_x \Leftarrow c_x]$. Thus, substitution is a reversible operation.

Another important property of substitution is that the order of substitution is not relevant.

LEMMA 8. (i) *Let* c *be a computation tree and let* x *and* y *be incomparable nodes, separable by* $<$. *Let* $c_x \equiv c_x'$ *and* $c_y \equiv c_y'$. *Then* y *is separable in* $c[c_x \Leftarrow c_x']$ *and*

$$c[c_x \Leftarrow c_x'][c_y \Leftarrow c_y'] = c[c_y \Leftarrow c_y'][c_x \Leftarrow c_x'] .$$

(ii) *Assume that* $y \in desc(x)$, x *is separable by* $<$, $c_x \equiv c_x'$, *and denote* $c[c_x \Leftarrow c_x']$ *by* $\bar{c}$. *Assume also that in* c_x' y *is separable and let* $(c_x')_y \equiv c_y'$. *Then* y *is separable in* $\bar{c}$ *and*

$$\bar{c}[\bar{c}_y \Leftarrow c_y'] = c[c_x \Leftarrow c_x'[(c_x')_y \Leftarrow c_y']] .$$ □

The notion of substitution can be generalized, to allow for substitution of a subforest of a computation by an equivalent forest. We say that a partial front M is *weakly separable* by $<$, if for each $y \in leaves(x) - leaves(M)$, either $y < leaves(M)$ or $leaves(M) < y$. That is, the steps of the computation under M are not interleaved with operations that are not under M. However, the subtrees rooted at M may be interleaved with each other. For a single node, separation and weak separation are the same.

Assume M is weakly separable by $<$. Using the same arguments as for a single separable node, we have that $c[S]$ is a composition of three atomic S-computations, the middle one being $c_M[S]$, where c_M is the subcomputation rooted at M. It follows that c_M is a computation forest. Assume that for all y, if $y < leaves(M)$ then $y < M$ and if $leaves(M) < y$ then $M < y$. Let d_M be a computation forest such that $c_M \equiv d_M$, and both have the same poset of roots and the same

return value from each root. Then $c[c_M \Leftarrow d_M]$ is well defined.

THEOREM 3' (Substitution Theorem) *For* c, M, c_M *and* d_M *as above,* $c' = c[c_M \Leftarrow d_M]$ *is a computation tree and* $c \equiv c'$. □

The results about reversibility of substitution and irrelevance of substitution order generalize in the obvious way. We note that Theorems 2 and 2' are special cases of the generalized substitution theorem. In these theorems, *reduction*, which is a special case of substitution, is used. Of course, *expansion*, the dual of reduction can also be used in these theorems.

As observed in the previous section, substitution by itself is not sufficient for proving correctness of computations, since we still need to find computations that can replace subcomputations. Tools are needed that will allow us to transform a computation to an equivalent one without relying on previously known equivalences. So far, the only tool available to us is the use of (C4).

COROLLARY 3. *Let* c *be a computation and* M *be a front that is separable by* $<$. *If each node in* M *is either a leaf or a node whose children are leaves, then* $c \equiv c/M$. □

Note that Corollary 1 follows directly from this result.

In our running example, the nodes r_1, w_1 and r_2 on level 2 are separable. To separate w_2, we will need to reverse the order of f_2^1 and f_1^1, relying on their commutativity. Then, by (C4), the computation becomes equivalent to a computation that contains only levels 0-2, with the order $w_2 < r_1 < w_1 < r_2$. Level 1 is still not separable.

7. CONFLICT PRESERVING SERIALIZABLE LOGS

In all practical applications, concurrency control algorithms rely on knowledge about conflicts between operations. When an incoming operation conflicts with an operation that is already executing, then either the new operation is delayed, or one of the transactions aborted. Thus conflicting operations are prevented from being concurrently executed. The theory that deals with these algorithms and their correctness will be presented now.

7.1 Ensuring Transaction Validity

By condition (C7), for every computation c, $c[\mathcal{S}]$ is an atomic $\mathcal{S}$-computation. Extending the order on the leaves to be serial does not invalidate (C5) and (C6), and the result is a computation c' equivalent to c and with the same root. From now on, we assume that $c[\mathcal{S}]$ is a serial $\mathcal{S}$-computation.

For a computation c with partial order $<$, we use $c[< \Leftarrow <']$ to denote the hierarchical tuple that results from replacing $<$ by $<'$. This tuple is not necessarily a computation.

The basic observation that is used in conflict based treatment of computations is that the order of commuting leaves may be changed (see Proposition 1). However, in a computation tree, we have to satisfy also property (C6), i.e., transaction validity.

LEMMA 9. *Let* $<'$ *be any partial order such that* $<_t \subseteq <'$. *Then each internal node* o *in* $c' = [< \Leftarrow <']$ *sees a* t(o)*-computation, i.e.,* c' *satisfies (C5).* □

Since $<$ is total on the leaves, for every two leaves x and y, either $x < y$ or $y < x$. We say that leaves x and y are *adjacent* if for every other leaf z, either $z < \{x,y\}$ or $\{x,y\} < z$.

Assume x and y are adjacent, say $x < y$. If $x \not<_t y$, then their order can be reversed without violating (C6). However, order-tree compatibility may no longer hold, so additional changes to $<$ may be required to restore the validity of (C5).

LEMMA 10. *Let* x,y *be adjacent leaves of* c *such that* $x < y$ *but* $x \not<_t y$, *and let*

$$<_1 = < - \{(z_1,z_2) \mid x \in desc(z_1), y \quad desc(z_2)\} \cup$$

$$\{(y,z) \mid z = x \text{ or } x < z\} \cup \{(z,x) \mid z < y\} .$$

Then $<_1$ *is a partial order and* $c_1 = c[< \Leftarrow <_1]$ *satisfies (C5) and (C6).* □

From now on, whenever we reverse the order of adjacent leaves that are not $<_t$-related, we assume that the appropriate changes to $<_1$ as described in the lemma, are made.

7.2 Tree Logs

Now, if x and y are adjacent leaves, there exists an intermediate state preceding x. If x and y commute w.r.t. this state, then their order can be reversed, the result being a computation that is equivalent to the one given. Algorithms that use state based commutativity potentially offer more concurrency. However, the need to use state specific information implies higher overhead and more complex algorithms. The algorithms in use today all use general commutativity, relying on suitably chosen conflict predicates. As we remarked previously, it is possible, by using a commutativity parameter, to come closer to state based commutativity without giving up the use of conflict predicate. The family of computations that are serializable by state based commutativity transformations is studied in the full paper but is not considered further here.

In the following, state specific information is not used, hence the states are omitted. We state our results in terms of operation commutativity, so return values also can be omitted. All our results remain valid for operation-f commutativity. However, then the f-values must be retained in the description of the computation, and some results need to be rephrased. The details are mostly left to the reader.

Assume that s_1, s_2, r are given. A *tree log* is a triple $\ell = (T_1, <, <_t)$ such that

$c = (s_1,T,<,r,s_2)$ is a computation tree and $<_t$ is a transaction order for c. We say that ℓ is *derived* from c. A log may be derived from several computations, but for the discussion we fix one of them. Note that our definition generalizes the concept of a log in classical serializability theory, where a log is a poset of operations, and the transaction order is assumed to be known (and contained in the given partial order). If a commutativity parameter f is used, then the log should contain also the f-values.

We will use freely for logs the terminology previously used for computations. Thus, we talk about *forest logs*. A tuple $(O,<)$ is an $\mathcal{S}$-*log* if it is derived from an $\mathcal{S}$-computation; in particular, $\ell[\mathcal{S}]$ is an $\mathcal{S}$-log. Logs ℓ_1 and ℓ_2 are *equivalent*, $\ell_1 \equiv \ell_2$, if $c_1 \equiv c_2$. (Note that the *same* s_1, s_2, r must be used whenever several logs are discussed.)

7.3 C-Separability and GCPSR Logs

Let $k = (O,<)$ be a serial $\mathcal{S}$-log and let CON be a symmetric binary relation on OP. We say that CON is a *conflict predicate for* k if the following property holds.

(C8) For every total order $<'$ on O that extends $< \cap$ CON, $(O,<')$ is an $\mathcal{S}$-log (w.r.t. same states and return value function).

The definition carries over in the obvious way to tree logs. CON is a conflict predicate for $\ell = (T,<,<_t)$ if it is a conflict predicate for $\ell[\mathcal{S}] = (\text{leaves}(T),<[\text{leaves}(T)])$. The *conflict order* defined by CON on ℓ is $<_c^{CON} = <[\text{leaves}] \cap$ CON. Note that $<_c^{CON}$ is acyclic but not necessarily transitive. We refer to it as an order, since (C8) holds for it iff it holds for its transitive closure.

LEMMA 11. *Every general conflict predicate is a conflict predicate for all logs.*

Proof. The lemma is an immediate corollary of Proposition 1. □

The converse of the lemma does not necessarily hold. A conflict predicate for a log ℓ may be constructed using information specific to ℓ, and it may indicate that x and y commute in ℓ even though they conflict w.r.t. some state.

A triple $\ell = (T,<,<_t)$, where $<_t \subseteq <$, is called a *representative log* if it satisfies the following property

(C9) For every order $<'$ that contains $<$ and is total on leaves(T), $\ell[< \Leftarrow <']$ is a tree log (and $<_t$ is its transaction order).

The given triple ℓ is *not* required to be a tree log.

Let CON be a conflict predicate for $\ell = (T,<,<_t)$. The *essential order* defined by CON on ℓ, denoted $<_e^{CON}$, is $(<_c^{CON} \cup <_t[\text{leaves}(T)])^+$. Intuitively, the order of leaves that are either $<_c^{CON}$-related or $<_t$-related cannot be changed. The essential order captures this information. We note that $<_e^{CON} \subseteq <$, hence it is a partial order on leaves(T), and $<_e^{CON}\uparrow$ contains $<_t$.

PROPOSITION 3. *Let ℓ and* CON *be as above. Then $\ell[< \Leftarrow <_e^{CON}\uparrow]$ is a representative log. Conversely, if $\ell[< \Leftarrow <']$ is a representative log, than there exists* CON, *a conflict predicate for ℓ, such that $<_e^{CON} = <'$.*

Sketch of Proof. Since $<_e^{CON}$ contains $<_t$ and $<_c^{CON}$, we can use Lemma 10 and (C8) to show that (C6) and (C7) are satisfied when $<_e^{CON}$ is replaced by any tree compatible $<'$ that is total on leaves(T). □

The advantage of a representative log is that it represents a collection of logs, differing only in their orders, and that testing if one of these orders separates M is easy.

LEMMA 12. *A total order $<$ separates a partial front M iff it separates $\hat{M} = M \cup (\text{leaves}(T) - \text{leaves}(M))$.* □

From now on, we restrict our discussion to fronts. We say that a front M is C-*separable* by $<_e$, where $<_e$ is a partial order on leaves(T), if there exists a total extension $<$ of $<_e$ that separates M.

PROPOSITION 4. *The following are equivalent*

(1) M *is* C-*separable by* $<_e$.

(2) $(\tilde{<}_e\uparrow)[M]$ *is acyclic.*

Proof. (1) ⇒ (2) Let $<$ be a total extension of $<_e$ that separates M. Then, by Corollary 2, $(\tilde{<}\uparrow)[M] = (<\uparrow)[M]$ is a total order on M. Since $<_e \subseteq <$, it follows that $(\tilde{<}_e\uparrow)[M]$ is acyclic.

(2) ⇒ (1): Since $(\tilde{<}_e\uparrow)[M]$ is acyclic, it can be extended to a total order on M, say $<'$. Then $<_e \subseteq (<'\downarrow)[\text{leaves}(T)]$ and the latter separates M, hence so does each of its total extensions. □

COROLLARY 4. *Let $\ell = (T,<,<_t)$ be a log, let* CON *be a conflict predicate for ℓ, and M be a front. If $(\tilde{<}_e^{CON}\uparrow)[M]$ is acyclic, then there exists a total order $<'$ that contains $<_e^{CON}$ and separates M, and $\ell \equiv \ell' = \ell[< \Leftarrow <']$.* □

Viewing the nodes of M as transactions, and the leaves under them as their operations, C-separability means that these transactions can be serialized without changing the order of conflicting operations or the transaction order. Thus, Proposition 4 generalizes the well-known result [BSW,Pa] that a log is CPSR iff its conflict graph is acyclic. The generalization is in that there only the conflict order is used. We use here $<_e$ which combines conflict order and transaction order.

A log $\ell = (T,<,<_t)$ is *(general) conflict preserving serializable*, abbr. CPSR (GCPSR), if it satisfies one of the following conditions.

(1) It is a log of depth 1.

(2) There exists a conflict predicate for ℓ (a general conflict predicate), say CON, and a nontrivial front M (i.e., a front that is not the root or the leaves) such that M is C-separable by $<_e^{CON}$ and, for some separating order $<'$ that contains $<_e^{CON}$,

(i) Every sublog of $\ell[< \Leftarrow <']/M$ is CPSR (GCPSR), and

(ii) The remainder log $\ell[< \Leftarrow <']/M$ is CPSR (GCPSR).

A computation is (G)CPSR if the log derived from it is (G)CPSR.

LEMMA 12. Every GCPSR log is CPSR.

THEOREM 5. Every CPSR log is derived from a serializable computation. That is, CPSR $\subseteq$ SR.

Proof. The proof is by induction on the depth of the log. The basis, where the depth is 1, is obvious. Assume then that the claim is true for logs of depth $\leq n$, and let ℓ be of depth n+1. Denote the corresponding computation by c. By (2i) in the CPSR definition, let $<'$ be a separating order. Then $\ell \equiv \ell' = \ell[< \Leftarrow <']$, since they have the same states and return values. By induction hypothesis, each sublog of ℓ' rooted at some $x \in M$ is serializable hence can be replaced by its root x. Thus, $\ell' \equiv \ell'/M$. Applying the induction hypothesis once more, ℓ'/M is serializable, hence so is ℓ. □

In our running example, let us use the standard conflict predicate for read and writes for the leaves on level 3. Then level 2 is C-separable, the separating order being $f_2^1 < s_2 < f_1^1 < f_1^2 < s_1 < f_2^2$. The order on the leaves of the remainder log is $w_2 < r_1 < w_1 < r_2$. Using again the standard conflict predicate, the front on level 1 is also C-separable. Since all sublogs are of depth 1, this log is GCPSR. Additional examples will be presented later.

The theorem induces a recursive technique for proving serializability. Beside the management of sub-problems, we only need to test for C-separability and find separating orders. Both are easy when CON and M are given. The techniques generalizes CPSR theory for flat logs, hence we are able to use results from this theory, as shown in the examples.

In the full paper we show that reduction of a tree can always be done bottom up, from the leaves to the root. That is, the technique can be used iteratively, instead of recursively. We also present examples that indicate that choosing the order of reduction may be a nontrivial problem.

6. CONFLICT PREDICATES FOR SUBLOGS AND REMAINDER LOGS

The definition of GPCSR logs does not require the same conflict predicate to be used. We now discuss the relationships between conflict predicates, conflict orders and essential orders of a log, its sublogs and the remainder log.

In what follows, let $\ell_0 = (T, <_0, <_t)$ be a given log, CON a conflict predicate for ℓ_0 and M a front that is C-separable by $<_e^{CON}$. To simplify the notation, we use $<_c$ and $<_e$ for $<_c^{CON}$, $<_e^{CON}$, resp. Also, we write logs with a partial order deifned on leaves(T), assuming implicitly that each such partial order is pulled up to a partial order on T. Thus, $\bar{\ell} = (T, <_e, <_t)$ is the representative log. We use $<$ to denote a total order on leaves(T) that extends $<_e$ and separates M, and we write $\ell = \bar{\ell}[<_e \Leftarrow <]$ for the log it defines.

6.1 Sublogs

For each $x \in M$, let $CON^x = CON[\text{leaves}(x)]$ and $CON/M = CON - \cup_{x \in M} CON^x$, that is, CON/M relates only pairs from distinct subtrees rooted at M. For a partial order d (in particular, for $<_t, <_c, <_e$ or $<$) we write $<_d^x$ for $<_d[\text{leaves}(x)]$ and $<_d/M$ for $<_d - \cup_{x \in M} <_d^x$.

LEMMA 13. *$<_d \subseteq <_f$ iff, for all $x \in M$ $<_d^x \subset <_f^x$, and $<_d/M \subseteq <_f/M$.* □

COROLLARY 5. *$<$ is an order that extends $<_e$ and separates M iff, for all $x \in M$ $<^x$ extends $<_e^x$ and $</M$ is the pull down of a total order on M that extends $(\tilde{<}_e\uparrow)[M]$.*

Proof. By the lemma, $<$ extend $<_e$ iff for all x $<^x$ extends $<_e^x$ and $</M$ extends $<_e/M$. In addition, $<$ separates M iff $(<\uparrow)[M]$ is a total order on M that extends $(\tilde{<}_e\uparrow)[M]$.

It is easily seen that $</M = ((<\uparrow)[M])\downarrow$ and if $(<\uparrow)[M]$ extends $(\tilde{<}_e\uparrow)[M]$ then $</M$ extends $<_e$. □

What we have just shown is that a separating order $<$ is a union of distinct components, each of which can be chosen independently of the others. The component $<^x$ is chosen from the set of total orders on leaves(x) represented by $<_e^x$. The component $</M$ is determined by choosing one of the total orders on M represented by $(\tilde{<}_e\uparrow)[M]$. It is this last component that actually separates M. Note that $(\tilde{<}_e\uparrow)[M] = (\tilde{<}_e/M)\uparrow[M]$. Thus, only the $<_e$-relationships among leaves of different subtrees rooted at M determine whether M is C-separable. Conflicts and timing constraints within a subtree are irrelevant. In other words, only CON/M is relevant for C-separability, and M is C-separable if $(<_e/M)\uparrow$ is acyclic on M.

For $x \in M$, ℓ_x is a tree log, derived from c_x, where c is the computation from which ℓ is derived. It is easy to see, using Lemma 4, that $<_t^x$ is the transaction order of this log. Let us consider conflict predicates for ℓ_x. The obvious choice is CON^x.

COROLLARY 6. *CON^x is a conflict predicate for ℓ_x, and $<_c^x$, $<_e^x$ are the conflict order and*

essential order, respectively, defined by it for ℓ_x. □

COROLLARY 7. *Let* $\bar{\ell} = (T, <_e, <_t)$ *Be a representative log, defined by* CON, *and let* x *be* C-*separated by* $<_e$. Then $\bar{\ell}_x = (T_x, <_e^x, <_t^x)$ *is a representative log, representing all (and only) the sublogs rooted at* x *of the logs corresponding to separating orders for* x *that extend* $<_e$. □

We note that for different separating orders, the initial and final states of the subcomputations rooted x may be different. Thus, $\bar{\ell}_x$ can possibly be associated with several state pairs. This is of no significance, since the states are not used.

By the corollary there is no need to compute a separating order; the relevant orders can be obtained by restricting $<_c$ and $<_e$ to leaves(x).

If we want to use another conflict predicate, say $\overline{CON}^x$, we need to compute explicitly a separating order, and use it to compute the new conflict and essential order. We note that, in general, the replacement of CON^x in CON by $\overline{CON}^x$ is not necessarily a conflict predicate for ℓ_0, (though we expect that in most practical applications it will be).

PROPOSITION 5. *Let* $\overline{CON}^x$ *be a conflict predicate for* ℓ_x, where $\ell = (T, <, <_t)$, $<$ *is a separating order for* M, *and* $x \in M$. *Then*

$$<_c^{\overline{CON}^x} = (<_e^x \cap \overline{CON}^x) \cup (<^x - <_e^x) \cap \overline{CON}^x .$$

Proof. The claim follows easily from the fact that $<_e^x \subseteq <^x$. □

8.2 The Remainder Log

We turn now to consider the remainder log $\ell/M = (T/M, (<\uparrow)[M], <_t[M])$. Since the nodes of M are not necessarily leaves of ℓ, we have to use a new conflict predicate. Let CON^M be a conflict predicate for ℓ/M, and let $<_c^M$ and $<_e^M$ be the conflict and essential predicates defined by it for ℓ/M. Even though $<_c^M$ and $<_e^M$ are determined by CON^M, they are related to $<_c$ and $<_e$ by the fact that $<$ is constrained to contain $<_e$.

PROPOSITION 6. (1) *Let* $<_d^M$ *be any one of the four relations* $(\lesssim_c\uparrow)[M]$, $(\lesssim_c\uparrow)^+[M]$, $(\lesssim_e\uparrow)[M]$, $(\lesssim_e\uparrow)^+[M]$, *then*

$$<_c^M = (<_d^M \cap CON^M) \cup (((<\uparrow)[M] - <_d^M) \cap CON^M) .$$

(2) *Assume* CON^M *is a conflict predicate for the remainder logs obtained from all separating orders that contain* $<_e$. *Then every relation* $\bar{<}_c^M$ *on* M *that relates all and only pairs of* $CON^M[M]$, *and such that* $\bar{<}_c^M \cup (\lesssim_e\uparrow)[M]$ *is acyclic, is the conflict order defined by* CON^M *for the remainder log of some separating order that extends* $<_e$.

Proof. (1) For each of the four possibilities for the value of $<_d^M$, $<_d^M \subseteq (<\uparrow)[M]$. The claim follows.

(2) By the given properties, $\bar{<}_e^M$ can be extended to a total order $<^M$ on M, that contains $(\lesssim_e\uparrow)[M]$. Thus, $<^M = (</M)\uparrow$ for some separating order that extends $<_e$. The claim follows. □

The first part of the proposition brings into focus two cases that need to be considered in the relationship between CON, defined on the leaves, and CON^M, defined on M. Let $x,y \in M$, and let p range over leaves(x), q range over leaves(y).

Case 1: x and y commute (w.r.t. CON^M), but for some p and q, p and q do not commute (w.r.t. CON). Given a total order on leaves(T), p and q are $<_c$-related, say $p <_c q$. Hence, $x \lesssim_c\uparrow y$. However, this relationship does not appear in $<_c^M$. As an example in a banking system, a transfer commutes with an audit, even though their accesses to individual accounts do not commute.

Case 2: For all p and q, p and q commute, hence x and y are not $\lesssim_c\uparrow$-related. However, it is possible that $(x,y) \in CON^M$ and a $<_c^M$-relationship between them needs to be chosen. As an (somewhat contrived) example, assume the existence of an operation that reads one of several counters, returns the value read and increments the counter. Two such operations conflict in general. Yet, in a given log, if they read different counters, their leaves commute.

The second part of the proposition allows us to compute a new $<_c^M$ by (strongly) pulling up $<_e$, and extending this pull up in an arbitrary way to an acyclic relation that covers all pairs of CON^M. Note that if $\lesssim_c\uparrow$ is used instead, than an arbitrary extension will not do; the extension must be compatible with $<_t$ on M. Also note that the assumption that CON^M is a conflict predicate for all separating orders is necessary since we can generate any one of these orders.

Let us now consider specific choices for CON^M. The obvious choice is to use a general conflict predicate CON_0^M. I.e., if x and y commute w.r.t. CON_0^M then they commute in general. The fact that CON_0^M is a conflict predicate for $(T/M, <_t[M])$ is a corollary of Lemma 11. Note that in the relationship between CON_0^M and CON both cases described above can occur. Hence, CON_0^M defines different conflict and essential orders for different separating orders. For a separating order $<$, these will be denoted by $<_{c,0}^M$ and $<_{e,0}^M$, resp. They can be computed using (1) of Proposition 7.

Consider Case 2 above. Intuitively, it would seem that if, in the given log, the leaves under x and y commute, then x and y should not be considered to conflict. We formalize this as follows. Let $\ell = (T, <, <_t)$ be a log such that $<$ is total on leaves(T) and separates M. For any total order $\bar{<}^M$ on M, let $\bar{<}$ be $(\bar{<}^M\downarrow)[\text{leaves}(T)] \cup (\bigcup_{x \in M} <^x)$. Obviously, $\bar{<}$ is a total order on leaves(T) that also separates M. It is obtained from $<$ by changing the relative order of sublogs rooted at M, without changing the order within any of the sublogs. We say that CON^M is *a conflict predicate for* ℓ/M *in* ℓ if the following property holds.

(C10) For every total order $\tilde{<}^M$ on M that extends $(<\uparrow)[M] \cap COM^M$, $(leave(T), \tilde{<})$ is an $\mathcal{S}$-log (w.r.t. the same states and return values as in ℓ).

LEMMA 14. *Using the previous notation, if ℓ_x is reducible to a single node for each $x \in M$, then a conflict predicate for ℓ/M in ℓ is a conflict predicate for ℓ/M.*

Proof. We have to show that under the given conditions, (C10) implies (C8). Now, given $\tilde{<}^M$, $\tilde{<}$ defines a serial $\mathcal{S}$-log on leaves(T) and separates M. The sublog consisting of leaves(x) can be replaced by x, for each $x \in M$, hence $\tilde{<}^M$ defines a serial $\mathcal{S}$-log on M. □

Given CON, a conflict predicate for ℓ, define $CON_1^M = \{(x,y) \mid x,y \in M, (leaves(x) \times leaves(y)) \cap CON \neq \emptyset\}$. I.e., CON_1^M is the set of pairs of elements of M that have a pair of noncommuting leaves under them. We refer to this type of conflict (commutativity) as leaf-based conflict (commutativity).

As we have seen, if CON is used as a conflict predicate for sublogs, then for all separating orders, the representative sublog rooted at x is $\bar{\ell}_x$. We thus obtain.

COROLLARY 7. *If, for each $x \in M$, $\bar{\ell}x$ is CPSR, then CON_1^M is a conflict predicate for $\bar{\ell}[<_e \Leftarrow <]/M$, where $<$ is any order that separates $<_M$ and extends $<_e$* □

Note that the use of leaf conflicts relies on the same idea as the use of a commutativity parameter, to obtain additional information from the computation. However, leaf conflicts are log specific, hence are not definable by commutativity parameters.

Denote the conflict and essential orders defined by CON_1^M on M for a separating order $<$ by $<^M_{c,1}$ and $<^M_{e,1}$, resp. They can be computed by (1) of Proposition 7. However, we note that, by definition of CON_1^M, Case 2 above does not apply.

PROPOSITION 7. (1) $<^M_{c,1} = (\tilde{<}_c\uparrow)[M]$.

(2) $(\tilde{<}_e\uparrow)[M] \subseteq <^M_{e,1} = (\tilde{<}_e\uparrow)^+[M]$. □

COROLLARY 8. *CON_1^M generates the same conflict and essential orders, hence the same representative log, for all separating orders.* □

It would seem advantageous to combine leaf commutativity with regular commutativity. Let CON_2^M be $CON_0^M \cap CON_1^M$, i.e., x and y conflict w.r.t. CON_2^M only if they generally conflict, and for some $p \in leaves(x)$, $q \in leaves(y)$, p and q also conflict. It turns out that CON_2^M is not necessarily a conflict predicate for ℓ/M.

Example 2. Consider a system with a fixed set of variables, x,y,z,..., distributed among two nodes A and B. In a consistent state, each variable resides at precisely one node. A read(x) operation tries to read x from A; if A is not there, it reads x from B. A write(x) uses a similar protocol, but it searches in B first. A move(x,source,dest), reads x's value from source, writes it into dest, then deletes x from source.

A sample log is shown in Fig. 2. The leaves are ordered in time order from left to right. The conflicts on the leaves are the pairs $(w_1(x,A), r_2(x,A))$, $(w_3(y,B), r_1(y,B))$ and $(r_2(y,A), w_3(y,a))$. Assume the only $<_t$ relationships are among the leaves under a node of level 2, e.g., $r_1(x,A) <_t w_1(x,A)$. Thus, level 2 is C-separable. The separating order is obtained by moving $r_2(y,A)$ to the left of $r_3(y,A)$, and the pair $r_1(y,B), w_1(y,B)$ to the right of $w_3(y,A)$. The pull up of the separating order to level 2 is $w_1(x) < r_2(x) < r_2(y) < move_3(y,A,B) < w_1(y)$. Using CON_2^M as a conflict predicate, the only pair in conflict is $(w_1(x), r_2(x))$. For any other pair, either the operations commute, or the leaves under them commute. For example, move commutes with every other operation; $r_2(y)$ leaf commutes with $w_1(y)$. Thus, using CON_2^M we obtain that level 1 is C-separable, implying that the log is serializable. This contradicts the fact that t_2 reads x from t_1, but reads a y-value that existed before t_1 wrote into y. □

To understand the error, consider the sequence $r_2 < move_3 < w_1$, that needs to be transformed into, say, $move_3 < w_1 < r_2$. In this sequence, r_2 and $move_3$ commute, so we can reverse the order of r_2 and $move_3$, obtaining $move_3 < r_2 < w_1$. Now we want to use leaf commutativity of r_2 and w_1 to reverse their order. This cannot be done however. The translation of r_2 as given in the log is valid only for $r_2 < move_3$. If r_2 follows $move_3$, its translation is different, namely, $r_2(y)$ reads y from B. For the new translation, r_2 and w_1 do not leaf commute.

In the full paper we show how the two types of conflict predicates can be combined in a restricted way.

9. EXAMPLES

We now present three examples of the use of our results. Moss's [Mo] nested transaction model was one of the first to deal with the concept of nesting. Although it uses a well-known locking policy, no formal proof of its correctness has appeared. We present a simple proof of its concurrent correctness. We do not deal with the recovery aspects. However, under suitable semantics for the commitand abort operations, these can be integrated easily into our proof to show that concurrent correctness still holds.

Our second example is multiversion algorithms. We show that a satisfactory and rather simple treatment of these algorithms can be made in our framework, with no recourse to special theories, as developed, e.g., in [BG,82].

Our last example deals with the specification and implementation of an abstract data type that supports atomicity of its own operations and of transactions using them. Here, our framework is not sufficient. We need to use two additional concepts that have not been treated in the paper.

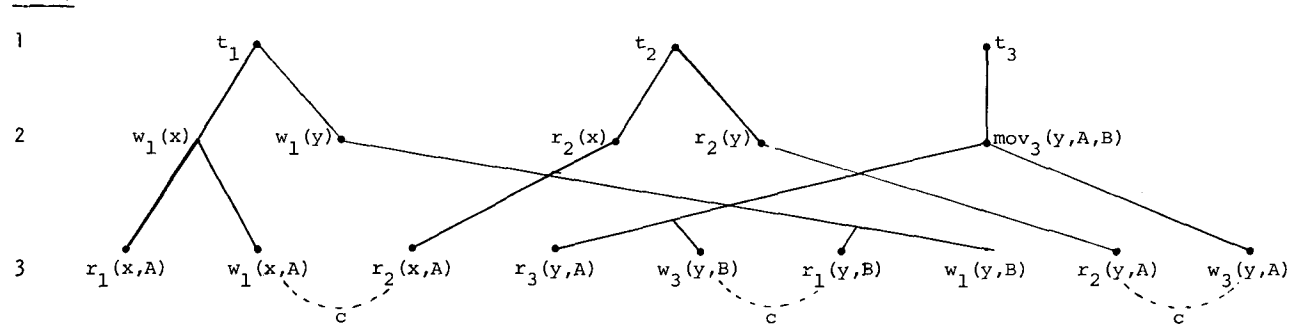

Figure 2. A sample log for a two page system.

These concepts seem to be of great significance for the treatment of concurrency in data types and multi-level systems. Their integration into our framework will be carried out elsewhere.

9.1 Moss's Algorithm for Nested Transactions

Moss [Mo] has suggested the following protocol for a general nested transaction system. For each node in the transaction tree, its subtransactions acquire and release locks according to the two phase locking (2PL) protocol, that is, a transaction cannot request a lock after releasing any lock. Locks released by subtransactions are retained by their parent. They can be acquired by other subtransactions under that parent, but not by any other transactions. After the parent releases any lock, none of its descendents can request a new lock.

Note that transactions are arbitrary. We regard transactions as being in conflict iff a pair of leaves under them is in conflict. Thus, leaf commutativity is used here.

To prove the correctness of the algorithm, let us consider the nodes on an arbitrary front as transactions, and the leaves under them as operations. It is easy to see that these transaction use the 2PL protocol w.r.t. each other. Each of them requests an appropriate lock before it performs an operation that may conflict with other operations. Each of them holds its locks (though it may pass them among its descendents) until it needs no new locks.

To show that a front M is C-separable, we have to show that $\tilde{<}_e\uparrow$ is acyclic on it, where $<_e = (<_c \cup <_t)^+$. Now, $(\tilde{<}_e\uparrow)[M] = (\tilde{<}_c\uparrow[M] \cup <_t[M])^+$, so it suffices to show the existence of a total order on M that extends $(\tilde{<}_c\uparrow)[M] \cup <_t[M]$. Further, since $<_t$ can be any suborder of the execution order $<$, we actually need to show that such an extension exists for $(\tilde{<}_c\uparrow)[M] \cup < [M]$.

We rely on a property of 2PL, proved in [BSW]. There, a flat log is called *strict serializable*, abbr., SSR, if it can be serialized without reversing the order of noninterleaved transactions.

THEOREM [BSW]. *If all the transactions use the* 2PL *protocol, then every (flat) log is* SSR. □

COROLLARY 9. *If all transactions obey Moss's protocol, then every tree log is* GCPSR.

Proof. The theorem guarantees that every front M is C-separable, since $(\tilde{<}_e\uparrow)[M]$ can be extended to a total order. For sublogs rooted at the front, the new conflict and essential order are $<_c^x$, $<_e^x$. For the remainder log, the new conflict and essential order are $(\tilde{<}_c\uparrow)[M]$ and $(\tilde{<}_e\uparrow)^+[M]$. These logs can be decomposed, using the same argument. The claim follows by induction. □

9.2 Multiversion Timestamping

In a multiversion database, each write on x produces a version of x; a read on x returns the value of one of the versions. Multiversion concurrency control algorithms have been described in [BEHR, BHR, CFLNR, Du, Re, Si, SR]. A theoretical treatment can be found in [BG82b, PK]. In particular, [BG 82b] presents a serializability theory specially tailored for multiversion algorithms. We present here a correctness proof for centralized multiversion timestamping, as first presented in [Re]. Our proof is simple, and it does not use the special theory developed in [BG 82].

In the algorithm, each transaction t_i obtains a timestamp, abbr. ts(i), as its first step. Timestamps are unique and are issued in a monotonically increasing sequence. Each version x_i of an item x has a write timestamp and a read timestamp, denoted $wts(x_i)$ and $rts(x_i)$, resp. It is always the case that $wts(x_i) \leqslant rts(x_i)$.

The operation $read_i(x)$, on behalf of transaction t_i, returns the version x_j of x such that $wts(x_j)$ is the largest wts of any version of x that is smaller than ts(i). Also, $rts(x_j)$ is replaced by $max(rts(x_j), ts(i))$. For a $write_i(x)$, two cases can occur. If, for some x_j, $wts(x_j) < ts(i) < rts(x_j)$, then the write is rejected and t_i is aborted, for the write would invalidate the value x_j returned by the read that created the current $rts(x_j)$. If no such x_j exists, a new version x_i, with $wts(x_i) = rts(x_i) = ts(i)$, is created.

While the transactions view the database as containing logical data items, the actual database contains many physical versions of a logical data item. Thus, we have a multilevel system, where operations on logical data items are executed as transaction on the actual system. A forest has

three levels. The transactions are found on level 1. The logical level operations are on level 2. These are get-ts(i), which returns a timestamp for t_i, $r_i(x)$ and $w_i(x)$. On level 3 we find the translations of $r_i(x)$ and $w_i(x)$. An $r_i(x)$ translates into a pair of operations: $select_i(x)$, which returns a version number j and also updates $rts(x_j)$, followed by $r_i(x_j)$. A $w_i(x)$ is translated into $w_i(x_i)$. We assume that the operations of aborted transactions do not appear in a completed computation.

Serial executions are defined as in Section 2. It is obvious from the description above that in a serial execution only the last version of an item is ever read, so the database behaves as a one copy database. Note that the select operation does not appear in the description in [BG 82b], and this is why a special theory with conditions that guarantee one-copy behavior needs to be developed there.

We now show that every log generated by the algorithm is GCPSR. Formal properties of the logs are presented in [BG 82b]. An intuitive understanding should suffice here.

The conflicts on the leaves are: $r_i(x_j)$ conflicts with $w_j(x_j)$, but not with any other $w_k(x_k)$ or $w_j(y_j)$. $Select_i(x)$ that returns j conflicts with any $w_k(x_k)$ such that $ts(j) \leqslant ts(k) < ts(i)$. The only such $w_k(x_k)$ that is allowed by the algorithm is when $k = j$. Note that operation-return value commutativity is used here. Finally, get-ts(i) conflicts with get-ts(j), but with no other operation.

We show now that the front containing the transactions, is C-separable. Assuming that $<_t$ relates only pairs of operations of the same transaction, there are no $\tilde{<}_t\uparrow$ links at level 1. The $<_c$ order on the leaves relates only get-ts(j) to get-ts(i), where $ts(j) < ts(i)$ and $w_j(x_j)$ to $select_i(x)$ and $r_i(x_j)$, where the select returns j. Here also $ts(j) < ts(i)$. It follows that $\tilde{<}_e\uparrow$ links at level 1 always lead from a low timestamp to a higher timestamp, hence there is no cycle and the front is C-separable.

Since the subtransactions of each transaction do not interleave, the sublog rooted at each t_i is serial, which shows that the log is GCPSR.

A final remark: The assumption that $select_i(x)$ and $w_k(x_k)$ are atomic is important. Assume a $select_i(x)$ selects the version x_j, but before it has updated its rts, a $w_k(x_k)$, where $ts(j) < ts(k) < ts(i)$, is allowed to execute. The w_k may find the old value of rts(j), hence complete successfully, instead of being rejected. This problem is avoided in practice by treating the $select_i$ and w_k as mutually exclusive sections.

9.3 Atomic Semi-Queues

Several papers [LiWe,ScSp] have recently considered the specification and implementation of atomic data types. The basic idea is that atomicity properties, like serializability and recoverability, have been so far implemented only in database systems. These papers advocate the construction of programming systems where arbitrary abstract data types and programs using them can be specified to have these properties.

Atomic semi-queues and their implementation are presented as an example in [LiWe]. A similar construction appears in [ScSp]. The goal is to ensure serializability of arbitrary transactions using semi-queues. In a regular queue, essentially only one transaction can be adding (removing) elements, for otherwise serializability cannot be ensured. In a situation where strict FIFO is not required, it is possible to weaken the FIFO requirement so as to obtain more concurrency.

A semi-queue is a set of cells containing values. Its operations are enq(x), which adds a cell with value x, and deq - a nondeterministic operation - that removes an arbitrary cell and returns its value. Deq is implemented to remove one of the oldest cells to ensure a measure of fairness, but this is irrelevant for concurrent correctness .

The proposed implementation is to use an extensible array of records. Each record contains a value and a flag with value e (enqueued) or d (dequeued). For record p, these are denoted p[val], p[flag]. The array can be dynamically extended at its high end; the operation enq(x) adds a record with contents (x,e). A deq is implemented by a locate, which searches the array from its low end, and returns the first record p such that p[flag] = e, followed by remove(p), which sets p[flag] to d and returns p[val]. If there is no p for which p[flag] = e, the locate waits.

The operations enq and locate are implemented as follows: A variable end points to the high end of the array. Enq is implemented by r(end) which returns p, where p is the highest record, followed by w(x,e,p+1) which writes (x,e) into the p+1'th record, followed by set(end,p+1). A locate loops through the array, starting at the low end, using access(p) to access the record p, and test-end(p) to test if p is (currently) the last record. We assume that the test is repeated until the value *false* is returned. Locate returns the first record p which is found to have p[flag] = e.

A generic operation tree appears in Fig. 3. A tree log has, of course, many operations of each type. Note that in a tree log, a transaction node may have many enq and deq children; similarly, a locate may have many access and test-end children. There is a unique child of each of the given types under an enq and a deq.

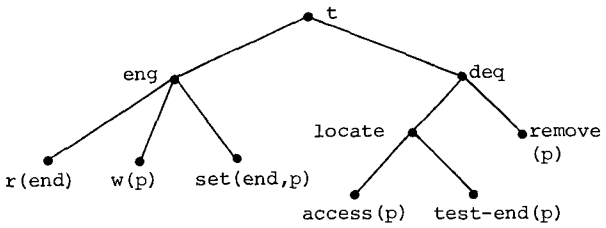

Figure 3. A Generic Operation Tree for Semi-Queues

The $<_t$ in a tree log can w.l.o.g. be assumed to be a total order on the children of each internal node. To describe the conflicts on the leaves, we use a notation where only the parameters of an operation that are relevant to conflicts are mentioned. Set(p) conflicts with every other set(q), with r and with test-end(p-1) and test-end(p); w(p) conflicts with every other w(p), with access(p) and remove(p); access(p) conflicts also with remove(p), and remove(p) conflicts with any other remove(p).

To ensure serializability, two locking protocols are used. First, each enq(x) locks end before its r(end) and releases it after its set(end,p). Thus, the enq's are made atomic with respect to each other. In our terms, enq's are separated from each other.

Consider now the interleaving of enq's and locates. Assume a locale returns p, and an enq increments end from q to q+1. If $p \leq q$, then there is no conflict among their leaves, so they can be separated. Assume $p > q$, i.e., the locate contains a sequence of $n \geq 0$ test-end(q) that return *false*, executed before the enq, followed by one test-enq(q) that returns true, executed after the enq. This sequence is equivalent to its last element, performed after the enq. We replace the sequence by this last element and thus separate the enq and locate, forcing the locate to follow the enq in the separating order. Note that if the locate also contains a test-end(q+1), it also follows the enq.

Now, nodes under locates do not conflict with each other. A remove(p) conflicts with at most one node, access(p), under a locate. Hence, we have shown that the front containing operations from {enq,locate,remove} is C-separable. The subtrees rooted at this front are of depth ≤ 1, hence can be pruned using (C4).

To determine the conflicts on the remainder tree, we return to our original view of the semi-queue as a set of cells, since we have essentially shown that the operations for this view, namely enq, locate and remove are executed atomically. Two enqs add different cells, hence they commute. To determine conflicts between enqs and locates, we use a commutativity parameter (see Section 2.1) with two components--the cell affected by an operation and the return value. An enq affects a cell p and returns $\perp$; a locate affects no cell and returns q. A conflict exists iff $p = q$. In this case, the locate follows the enq in the execution order and also in the pull up of the separating order. Similarly, an enq conflicts with remove(p) only if the enq affects p; such a remove follows the locate mentioned above.

The only problem that can prevent separation of the next front is the existence of several locates that return p and remove(p)'s. The second protocol requires that to perform access(p), p should be locked. If the lock request fails, the locate is free to access another cell. If p[flag] = d, the lock is released; if p[flag] = e, the lock is retained until after the remove(p). Thus, different locate remove pairs affect different records. The front containing the enq's and deq's is thus C-separable. In the separating order $<$, $enq_i(x) <\!\!\uparrow deq_j$ only if deq_j removes the cell created by $enq_i(x)$. Two deq's never conflict.

To allow further reduction of the log, the previous locking protocol is extended. Each enq and deq locks the cell it affects, and the lock is held by the issuing transaction according to the 2PL policy. It is now straightforward to check that all logs are CPSR.

10. CONCLUSIONS AND FURTHER RESEARCH

We have presented in this paper a general model for nested transaction systems. The model is stated in abstract terms, and in addition to nesting it allows arbitrary transactions and operations. We have also presented some tools for proving the equivalence of computations, or the serializability of computations. In particular, we have generalized the well-known CPSR concept to computation trees and provided tools for proving that trees are CPSR.

Our model is very general so that our results can be applied directly to a wide variety of applications. In particular, our model can serve as a unified framework for dealing with concurrency in database systems. We have illustrated this by proving Moss's algorithm and multiversion time-stamping. Other applications, e.g., hierarchical locking and replicated data, can also be treated.

There is still more work to be done in making our model applicable to various types of systems, where special information is being used in concurrency control. As an example consider multi-level systems. Such systems have a fixed number of levels, hence the structure of computation trees is fixed and the fronts to be used in reducing such trees are known. Abstract data types are a special case of such systems. Among the issues to be treated is the fact that there exist different views of the system at different levels, as illustrated in the semi-queue example. This can be captured by the concept of *state equivalence*. Another issue that may arise is when certain operations can be applied only to certain states. The correctness proof then needs to show that they are applied only to such states.

Search structures are also a special case of multilevel systems. The semi-queue can be considered a search structure. Our correctness proof shed light on an issue that seems to be typical in such structures, namely that in an interleaved executions operations are performed that will not appear in any noninterleaved execution. To account for this, our definition of CPSR needs to be generalized to allow changes to the translation of operations.

We have shown that the concept of commutativity can be extended by considering commutativity parameters. This is, potentially, a very useful concept. The treatment of synchronization mechanisms in database systems is oriented towards read/write operations, with read sets and write sets as arguments. Synchronization is often discussed in terms of the data items involved, e.g., locks on data

items or timestamps on data items. However, the view that a lock is a flag associated with an operation-f value pair may be more promising for other applications, such as atomic data types.

Our results on some of the issues raised above will appear in forthcoming papers.

BIBLIOGRAPHY

[ABG] Attar, R., P.A. Bernstein, and N. Goodman, "Site initialization, Recovery, and Back-up in a Distributed Database Systems," *Proc. 6th Berkeley Workshop*, Feb. 1982, pp. 185-202.

[BHR] Bayer, R., H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Sys.* 5:2 (June 1980), pp. 139-156.

[BG 81] Bernstein, P.A., and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* 13, 2 (June 1981), pp. 185-221.

[BG 82a] Bernstein, P.A., and N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control," *Proc. 8th VLDB*, Sept. 1982, pp. 62-76.

[BG 82b] Bernstein, P.A., and N. Goodman, "Multiversion Concurrency Control-Theory and Algorithms," *Proc. of the ACM SIGACT-SIGOPS Conf. on Principles of Distributed Computation*, August 1982, Ottawa.

[BGL] Bernstein, P.A., N. Goodman, and M.Y. Lai, "Laying Phantoms to Rest (by Understanding the Interactions Between Schedulers and Translators in a Database System)," *Proc. 1981 IEEE COMPSAC Conf.*, Oct. 1981.

[BSW] Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. on Software Engineering*, SE-5, 3 (May 1979), 203-215.

[CFLNR] Chan, A., S. Fox, W.T. Lin, A. Nori, and D. Ries, "The Implementation of an Integrated Concurrency Control and Recovery Scheme," *Proc. 1982 ACM SIGMOD Conf.*, ACM, N.Y.

[Dubo] Dubourdieu, D.J., "Implementation of Distributed Transactions," *Proc. 1982 Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 81-94.

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

[FGL] Fischer, J.J., Griffeth, N.D., and N.A. Lynch, "Global States of a Distributed System," *Proc. 1st IEEE Annual Symp. on Reliability in Distributed Software and Database Systems*, 1981, pp. 31-38.

[Gr] Gray, J., "The Transaction Concept: Virtues and Limitations," *Proc. 7th International Conf. on Very Large Data Bases*, Cannes, Sept. 1981, pp. 144-154.

[KW] Kwong, Y.S., and Wood, D., "A New Method for Concurrency in B-trees," *IEEE Trans. Softw. Eng.* SE-8, 3 (May 1982), pp. 211-222.

[La] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, 21, 7 (July 1978), pp. 558-565.

[LBY] Lehman, P.L., and Bing Yao, S., "Efficient Locking for Concurrent Operations on B-Trees," *ACM TODS* 6:4 (Dec. 1981), pp. 650-670.

[LiWe] Liskov, B., and W. Weihl, "Specification and Implementation of Resilient, Atomic Data Types," Manuscript, MIT Laboratory of Computer Sciences, 1982.

[Ly] Lynch, N.A., "Concurrency Control for Resilient Nested Transactions," *Proc. 2nd SIGACT-SIGMOD Conf. on Principles of Database Systems*, Atlanta, March, 1983.

[MaPn] Manna, Z., and A. Pnueli, book in preparation, 1983.

[Mo] Moss, T.E.B., "Nested Transactions: An Approach to Reliable Distributed Computing," Ph.D. Thesis, MIT Laboratory for Computer Science, 1981.

[Pa] Papadimitriou, C.H., "Serializability of Concurrent Updates," *J. ACM* 26:4 (Oct. 1979), pp. 631-653.

[PK] Papadimitriou, C.H., and P.C. Kanellakis, "On Concurrency Control by Multiple Versions," *Proc. 1st ACM SIGACT-SIGMOD Conf. on Principles of Database Systems*, March 1982.

[RSL] Rosenkrantz, D.J., R.E. Stearns, and P.M. Lewis, "System Level Concurrency Control for Distributed Database Systems," *ACM TODS* 3:2 (June 1978), pp. 178-198.

[Re] Reed, D., "Naming and Synchronization in a Decentralized Computer System," Tech. Rep. MIT/LCS/TR-205, MIT, Dept. of Elec. Eng. and Computer Science, Sept. 1978.

[SiKe] Silberschatz, A., and Kedem, Z., "Consistency in Hierarchical Database Systems," *J. ACM* 27:1 (Jan. 1980), pp. 72-80.

[ScSp] Schwartz, P., and Spector, A., "Synchronizing Shared Abstract Types," Tech. Rep. CMU-CS-82-128, CMU Dept. of Computer Science, Sept. 1982.

[SLR] Stearns, R.E., P.M. Lewis II, and D.J. Rosenkrantz, "Concurrency Controls for Database Systems," *Proc. 17th Symp. on Foundations of Computer Science, IEEE*, N.Y., 1976, pp. 19-32.

[SR] Stearns, R.E., and D.J. Rosenkrantz, "Distributed Database Concurrency Controls Using Before-Values," *Proc. 1981 ACM-SIGMOD Conf.*, ACM, N.Y., pp. 74-83.