

A Concurrency Control Algorithm for an Open and Safe Nested Transaction Model: Formalization and Correctness

SANJAY KUMAR MADRIA, S. N. MAHESHWARI* AND B. CHANDRA[†]

Department of Computer Science

University of Missouri-Rolla

Rolla, Mo 65401, U.S.A.

E-mail: madrias@umr.edu

**Department of Computer Science*

Indian Institute of Technology, Hauz Khas

New Delhi 110021, India

E-mail: snm@cse.iitd.ernet.in

†Department of Mathematics

Indian Institute of Technology, Hauz Khas

New Delhi 110021, India

E-mail: bchandra@maths.iitd.ernet

In this paper, we formalize and prove the correctness of a concurrency control algorithm for an open and safe nested transaction using I/O automaton model. The model uses the notion of a recovery point subtransaction in the nested transaction tree. Our nested transaction model uses a prewrite operation before an actual write operation to increase the concurrency. It is termed "open and safe" as prewrites allow early reads (before database writes on disk) without cascading aborts. In our model we have also modeled the buffer management operations as nested transactions, and the concurrency control algorithm controls their executions. Non-access subtransactions, objects and the scheduler are modeled as I/O automata with the help of some pre and post conditions. These conditions capture the operational semantics and behavior of each automaton during the execution of transactions. While modeling we also take into account log activities, which occur during the execution of transactions. We also briefly sketch the recovery algorithm. The correctness proof shows that the concurrency control algorithm for our model is serially correct. Our proof makes use of assertional reasoning and provides many interesting invariant, thus gives a better understanding of our transaction model and the concurrency control algorithm. While proving correctness, we mapped our system to that of Moss's two phase locking system to show the relationships between the two algorithms.

Keywords: nested transactions, I/O automaton model, recovery, concurrency, two phase locking

1. INTRODUCTION

1.1 Overview of Nested Transaction Models

Closed Nested Transaction Model The theory of nested transactions [1, 2] allows the benefits of atomicity to be available within a transaction. In a nested transaction model

Received July 2, 2001; revised May 21, 2002; accepted July 11, 2002.
Communicated by Ming-Syan Chen.

[1], a subtransaction may contain operations to be performed concurrently, or operations that may be aborted independently of their invoking transaction. Such operations are considered to be subtransactions (children) of the original transaction. This parent-child relationship defines a nested transaction tree; such transactions are termed nested transactions [1]. Failure of subtransactions may result in invocation of alternate subtransactions that could replace the failed ones to accomplish the successful completion of the whole transaction. A child transaction has access to the data locked by its parent. It is atomic with respect to its parent and its siblings. It is serializable with its siblings. It becomes permanent only if its parent becomes permanent. If a parent aborts, all its descendants' effects are to be undone. Therefore, a child's scope is restricted to only its parent. Hence, this model is termed a closed nested transaction model. A parent commits only after all its children are terminated.

In the closed nested transaction model [1, 3, 4], a subtransaction's updates are not visible outside its parent and, therefore, availability is restricted. If the parent aborts, the subtransaction is also aborted. That is, aborts are handled by aborting the transactions of all underlying operations. In the *closed nested transaction model*, the availability is restricted as the scope of each subtransaction is restricted to only its parent. This forces a subtransaction to pass all its locks and versions of data objects updated to its parent on commit. The effect of a committed subtransaction is made permanent only when its top-level transaction commits.

Drawbacks In many applications, it is unacceptable that the work of a long-lived transaction (common in engineering design applications [5, 6]), is completely undone by using either of the above techniques in case transaction eventually fails at finishing stage. The current strategy forces short-lived transactions to wait to acquire their locks until top-level transactions commit and release their locks. Therefore, the model is not appropriate for the system that consists of long and short transactions.

Open Nested Transaction Model Nesting in transactions corresponds to the nesting of procedures or to the nesting of layers of data abstractions. In the open nested transaction model [7-9], a subtransaction's modifications are visible to other transactions at the same level of data abstraction as soon as it commits, even if its parent is still active. Hence, it provides more availability in comparison to the first model. However, later, if the parent aborts, a suitable compensatory operation is initiated to remove the effects of already committed subtransactions. Since basic (i.e., read and write) locks are released early and have possibly been acquired by other transactions, an abort has to take place in the form of an inverse or compensatory operation.

A related but more complex notion of nesting, emphasizing level of data abstraction is used in system R [10] and has been studied in a number of papers [8-10]. In System R, locking is applied twice, first, on tuples until EOT and then on pages for the scope of each tuple action. Since tuple actions can be regarded as subtransactions. Page locks are released before the commitment of the entire application transactions. This technique of long tuple locks and short page locks has been called open nested transaction. The page level locks guarantee serializability of tuple actions, thus ensuring that they can be considered to be elementary and allowing tuple level concurrency. However, nesting is restricted to two levels only and the recovery scheme is complicated. In [8] the system is

organized using multiple-levels of abstraction, with concurrency control performed separately at each level. The nesting in such systems correspond to levels of data abstraction and allow replacement of entire subtree of nesting activity by single action as well as the reordering of actions in a history. This makes their technique more complicated. To exploit layer specific semantics at each level of operation nesting, Weikum presented a multi-level transaction model [9] called open nested transaction model. The model takes into account the commutative properties of the semantics of operations at each level of data abstraction to achieve a higher degree of concurrency. If two operations at the same higher-level commute then their conflicting descendants at the same lower level are allowed to execute since they will not introduce any inconsistencies. In this model, a sub-transaction is allowed to release locks on finishing before the commit of higher-level transactions. In case a higher-level transaction aborts, the aborted transaction's effect is undone by compensatory transaction. This model has also been studied in the framework of object-oriented databases in [12, 13].

In the open nested transaction model, the increased concurrency is provided by exploiting the commutative properties of the semantics of operations at the same level of data abstraction. In such models, the leaf level locks are released early only if the semantics of the operations are known and the corresponding compensatory actions defined at each level.

Drawbacks The semantics of transactions may not be known and not all actions may be compensated (e.g., handing over a check). In real time situations, there are other classes of operations that cannot be undone. These are the operations that have an irreversible external effect, such as handing over huge amounts of money at an automatic teller machine (ATM). Such operations have to be deferred until the top-level transaction commits, which restrict availability, i.e., increases response time.

1.2 Motivation for Open and Safe Nested Transaction Model

Consider the part of nested transaction tree for fund transfer operation from a group of accounts to another account. In the transaction tree, let T_s be a transaction on whose behalf T_{s1} invokes various subtransactions to collect (access) funds from different accounts. Once T_{s1} is committed, T_s invokes T_{s2} which finally credits the funds collected into the other account. Suppose after a subtransaction T_w has withdrawn the entire amount T_s commits. If any transaction situated above T_s aborts, then it is desirable for the transaction to complete transaction revival successfully. This is because it is not possible to undo or correct the failed transaction's action by some compensatory actions. Another possibility is to delay the actual commit of T_s until its top-level transaction commits, which restricts availability. For example, a balance transaction has to wait until the commit of the top-level transaction.

Consider another scenario where, in the nested transaction tree, a subtransaction determines the success (commit) or failure (abort) of the top-level transaction. Suppose this nested transaction tree models various activities (modeled as subtransactions here) related to a business travel. Some of the activities are very crucial in determining the completion of the top-level activity. For example, the commit of "fund" and "visa" subtransactions will determine whether or not the travel transaction will commit. That is, these subtransactions commit will determine the commitment of the top-level transaction no

matter what may be the fate of other subtransactions in the transaction tree. Note that once the fund and visa subtransaction commits, its upper level (sub)transactions will be forced to commit, though some delay or restart may involve.

Our Objective There are two basic motivations behind our open and safe new nested transaction model [14]. First, it is desirable that long-lived transactions be able to release their locks before top-level transactions commit. Second, it may not be desirable or possible to undo or compensate the effects of one or more of the important committed descendants after the failure of a higher-level transaction due to an abort or a system crash. Our model allows some particular sub-transactions to release their locks before their ancestor transactions commit. This allows the other subtransactions to acquire required locks earlier. Our model handles situations where a committed lower level subtransaction's effect cannot be undone or compensated in case of a higher-level transaction's failure. It is possible that a transaction's semantics may be such that, beyond a certain point, either it cannot rollback entirely or its effect should not be compensated. These are some of the improvements over the other models discussed before.

In this paper we introduce an open and safe nested transaction model in the environment of normal read and write operations to remove the deficiencies stated earlier and to further improve availability. Our model supports inter- and intra-transaction concurrency. We assume that semantics of transactions at various levels of nesting are not known. Our nested transaction model can handle the situations where a committed lower level subtransaction's effect cannot be undone or compensated in case of a higher-level transaction's failure. A transaction's semantics may be such that, beyond a certain point, it cannot rollback entirely or its effect should not be lost. Our model allows some particular sub-transactions to release their locks before their ancestor transactions commit. This allows the other subtransactions to acquire required locks earlier.

We introduce the concept of a "recovery point subtransaction" of a top-level transaction in a nested transaction tree. It is essentially a subtransaction after the commit which its ancestors are not allowed to rollback. In other words, once the recovery point subtransaction of a top-level transaction has committed, all its superior transactions are forced to commit. In case it aborts, its ancestors can choose an alternate path to complete their execution. It says that recovery point subtransaction's commit, (e.g., mailing a check) is crucial for the commit of its ancestors. In case a superior transaction aborts or the system fails after the commit of its recovery point subtransaction, the failed transaction has to complete on system revival. Such a transaction execution permits a recovery point subtransaction to reveal its result to other transactions at any level of nesting before its superior transactions commit. A recovery point subtransaction's effect is made durable before its top-level transaction's commit. This results in relaxation of the isolation property of the transaction.

In our model, to avoid undo actions and the consequent cascading aborts as well as to increase the availability, we assume that each write issues a prewrite operation [14, 15] for the objects it intends to write. Each prewrite operation contains the value that a user visible transaction wants to write and precedes the associated final write. A prewrite operation actually does not change a data object's state but only announces the value the data object will have after the associated write is performed. In response to a read operation, each DM (Data Manager) returns the prewrite value, if any, otherwise it returns

the write value. The advantage of prewrite is that a read operation can get the value before a data object's state is changed. Hence this results in increasing the availability further with reduced execution time. Prewrite operations are particularly helpful in increasing availability in engineering design applications [5, 6] and in large software design projects [16] where transactions are generally large.

In our open and safe nested transaction model [14], a subtransaction that initiates different prewrite access subtransactions at the leaf level for different data objects is defined to be the recovery point subtransaction. These announced prewrite values are made visible to other subtransactions after the commit of the recovery point subtransaction (hence open). The prewrite subtransactions release their locks before their ancestors commit. Discarding some of the prewrites before the commit of the recovery point subtransaction will not introduce cascading aborts (hence safe) since the prewrite values are made visible only after the commit of the recovery point subtransaction.

Our presentation in this paper has a three-fold structure. First, we present our nested transaction model, and the concurrency control algorithm that controls the execution of nested transactions in our model. We discuss the nested transaction system of our model by building a nested transaction tree and giving the precise locking rules to control the concurrent execution of nested transactions.

Second, an effort is made to give a clear and reasonable description of our nested transaction concurrency control algorithm using the I/O automaton model. We believe that various complex distributed algorithms such as our nested transaction processing model would benefit from a more rigorous analysis within the framework of the I/O automaton model otherwise. The level of detail in the algorithm makes careful reasoning very difficult. The transaction automata, object automata and scheduler automaton are described using some pre and post conditions. These pre and post conditions capture the semantics and behavior of each component. They are also useful in understanding the interactions among various components that occur during normal execution. We feel that these specifications can also be used directly for the implementation of the system as we demonstrated in [17]. Thus, careful description of the nested transaction system and its correctness using I/O automaton model provides a logical approach of describing the components which are much easier to understand and to reason about the behavior of each component by means of simple properties and assertions.

We model our system as a generic system composed of transaction automata, generic object automata and a generic scheduler automaton. Each non-access subtransaction is represented by an automaton, as is each data object. Since access subtransactions to the same DM need to communicate, we associate a single generic object automaton with each DM rather than one with each access. The generic scheduler transmits requests to the appropriate recipient with arbitrary delay, allowing siblings to run concurrently, and makes decisions about commit or abort. It passes reports about completion back to parents and informs objects of the fate of transactions. It leaves the task of coping with concurrency and recovery to generic objects. We have used a generic object automaton to maintain locks and the value of data object. It delays operations until it is permitted to respond according to the locking rules. Each of these automata is specified with the help of some pre and post conditions. These pre and post conditions are used to prove the properties describing the behavior of the system. These conditions capture the operational semantics, locking rules and also take into account the logging of

auxiliary information required for system recovery. For example, logging of information into lock, transaction and dirty data object tables, etc, are also specified in the pre and post conditions. For this reason, we have also outlined a brief recovery algorithm in this paper.

Third, we give a precise correctness proof of our model. Our proof of correctness shows the relationship between our transaction model and its concurrency control algorithm and the model presented in [18]. We show that the schedule obtained by some well-defined reordering of some operations in our model has the same ordering as obtained by a schedule of [18]. This establishes the relationship between our model and the model presented in [18].

1.3 Related Work on the Proof of Correctness

Some related work on the formal correctness using an I/O automaton models, [19] are as given below. Most of these algorithms are discussed using an I/O automaton model and more details on some of these algorithms are available in [19].

- In [20] Lynch has presented a complete proof of the exclusive locking algorithm for nested transactions. Reed [21] has presented a multi-version timestamp concurrency control algorithm to provide nested transaction based data management. Different versions of an object keep track of the “history” of the object. A timestamp is associated with each version of an object and is used to allow concurrent execution of subtransactions. In [22] a formal analysis of the multi-version timestamp-based algorithm is given. The algorithm is proved to be correct by showing that the objects used in the algorithm are all static atomic. The atomicity theorem is used to show that if all the objects in the system are static atomic then the system guarantees atomicity. The proof techniques given are very general and can be applied to a large classes of systems including those where different data objects are implemented independently, and where type of objects can be used to obtain increased concurrency.
- Moss [1] has extended two phase locking with separate read/write locks to handle nesting. A formal version of this algorithm has appeared in [18]. In [23] the read-update locking algorithm [24] has been generalized and a new commutative locking algorithm has been introduced to handle nested transactions. The paper defines a local atomicity condition for data objects, called dynamic atomicity. The atomicity theorem is used to show that if all the objects in the system are dynamic atomic then the system guarantees atomicity. The dynamic atomicity provides modularity as it allows one to verify the implementations of individual objects independently. It also allows the use of different implementation techniques and different algorithms in different objects as long as all the objects are dynamic atomic.
- Fekete et al. [25] presented a serialization graph construction for nested transactions. The proof technique has the same form as in the classical theory; one must show that a graph having transactions for nodes and edges representing ordering between transactions is acyclic. The serialization graph model introduces the concept of visibility that hides the effects of a subtransaction from any other subtransaction until all its ancestors, up to the least common ancestor (l.c.a.) are committed. They have defined a new kind of serialization graph and proved that under certain assumptions,

the absence of cycles in this graph is a sufficient condition to ensure the serial correctness of a system. They have applied their technique to verify the correctness of Moss's read/write locking algorithm for nested transactions, and an undo logging algorithm that has not been previously proved for nested transaction system.

- The multi-granularity algorithm given in [26] has been extended to nested transaction systems in [27]. The algorithm considered file and record level granularity. The Correctness proof shows that there is a possibility mapping to the abstract algorithm for commutability-based locking given in [23]. The paper shows that some objects can use multi-granularity locking while others use Moss's two phase locking and some other user-defined dynamic atomic algorithms. Some more related work by the same authors on concurrency control using predicate locks appears in [28, 29].
- Gifford's basic quorum consensus algorithm for data replication [30] is generalized by Goldman in [31] to accommodate nested transactions and transaction aborts. The presentation separates the treatment of replication entirely from concurrency control which helps to simplify the reasoning. The paper shows that any correct concurrency control mechanism may be used on the copies, considered as separate objects; the whole system will then appear to be atomic and non-replicated. More recently, we have reported a virtual partition algorithm in a nested transaction environment and its correctness in [32], where we have shown that the technique of proving the correctness in [31] is not applicable to the Virtual Partition algorithm. We define the weaker correctness criteria for our algorithm, not reported in [19].
- Nested transactions have also been discussed in the context of B-Trees [33] and linear hash structures [34]. In [33], the notion of "strongly-serially correct" behavior has been defined and used as the correctness criterion. The drawbacks of [33] have been removed in [34]. In the algorithm, the locks have been considered at both key and vertex level. These locks have been implemented in nested transaction environment using Moss's two phase locking algorithm and the locking protocols of a linear hash structure algorithm with lock-coupling technique. The linear hash structure algorithm in nested transaction environment is proved to be "serially correct". We have also studied multi-level transactions in a linear hashing environment [35].
- A short version of open and safe nested transaction model and outline of its correctness has appeared in [36]. The recovery model of our open and safe nested transaction model and its proof of correctness using I/O automaton model has appeared in [37]. Therefore, in this paper we have omitted the detailed recovery algorithm. We do talk about overview of the recovery model as we use some recovery components that are necessary in modeling the concurrency control algorithm for the sake of uniformity. We use some log variables such as transaction table, lock table, various logs, etc., which basically model recovery aspect and are processed during normal transaction processing.

1.4 Paper Organization

Section 2 reports some preliminaries on I/O automaton model, nested transaction system and correctness. Section 3 gives the nested transaction tree structure for modeling our transaction model using I/O automaton model. In section 4, we discuss the concurrency control algorithm and overview of the recovery algorithm. In section 5, we discuss

the modeling of transactions using I/O automata. In section 6, we discuss some properties of our model. Section 7 discusses the details of correctness. Finally, we conclude in section 8.

2. PRELIMINARIES

2.1 I/O Automaton Model

An I/O automaton A as defined by Lynch, et al. [2] is a 5-tuple $(states(A), start(A), out(A), in(A), steps(A))$. $states(A)$ is the set of states of A , $start(A)$ is the set of starting states and subset of $states(A)$. $out(A)$ and $in(A)$ are the sets of output and input operations, respectively. $steps(A)$ is the transition relation of A , which is a set of triples of the form (s', π, s) where s and $s' \in states(A)$, $\pi \in in(A) \cup out(A)$, i.e., automaton changes its state from s' to s on operation π . An element of the transition is called a step of A .

The finite alternating sequence $s_0, \pi_1, s_1, \dots, \pi_n, s_n$ of states and operations of A is called an execution of A . A schedule of A is the subsequence of an execution of A consisting of only the operations of A .

A set of I/O automaton may be composed to create a system S such that the sets of output operations are disjoint. Thus, a state of the composed I/O automaton is a tuple of states, one for each component and the start states are tuples consisting of start states of the components. Let α be a schedule of a system with a component A , then $\alpha|A$ is the subsequence of α containing exactly the operations of A . Clearly, $\alpha|A$ is a schedule of A . The reverse holds by the Composition Lemma [2] which is formally stated as follows: Let σ' be a schedule of a system S and let $\sigma = \sigma'\pi$ where π is an output operation of component A . If $\sigma|A$ is a schedule of A , then σ is a schedule of S .

2.2 Nested Transaction System and Correctness

A nested transaction system is modeled by a four-tuple $(\tau, parent, O, V)$ where τ is a set of transaction names organized into a tree by the mapping $parent: \tau \rightarrow \tau$ where T_o acts as the root. The set O denotes the set of objects; it partitions the set of accesses, where each partition block contains accesses to the particular objects. V is the set of return values. We can relate two nested transaction systems as follows: A nested transaction system $P = (\tau_P, parent_P, O_P, V_P)$ is called a structural extension of the nested transaction system $Q = (\tau_Q, parent_Q, O_Q, V_Q)$ if $\tau_P \subseteq \tau_Q$, $O_P \subseteq O_Q$, respectively, $V_P = V_Q$, and $parent_P$, restricted to τ_Q , = $parent_Q$.

A nested transaction processing system is said to be modeled using the I/O automaton model when each component of the system, namely each non-access transaction, object and the scheduler are modeled as an automaton. Formulation of nested transaction systems as I/O automata permits precise correctness conditions to be satisfied by the algorithm. These correctness conditions can be stated at the transaction interface that does not contain explicit information about object representation.

The correctness of a transaction processing system is defined in terms of a serial execution of the same system. That is, it requires an execution of the same system to exist in which transactions run one at a time without interleaving steps of different transactions. Correctness is defined by first giving a separate specification of permissible serial

executions as seen by a user of the system, and then defining how executions of a transaction processing system must relate to this specification. The permissible serial execution for a transaction processing system is defined by introducing the notion of a scheduler, which executes transactions serially. Such systems, called serial systems, are not constrained by the issues of concurrency control, recovery and transaction aborts.

Formally, a schedule α of a system is serially correct for a transaction T if its projection on T , $\alpha|T$, is identical to $\beta|T$ for some serial schedule β [2]. In other words, T sees same things in α that it would see in some serial schedule. α is serially correct if it is serially correct for every non-orphan, non-access transaction.

The principal notion of correctness for a transaction processing system is that of serial correctness of the root transaction T_o of all finite schedules. This says that “the outside world” cannot distinguish between the given system and the serial system. A fairly strong and possibly interesting correctness condition is the serial correctness of all non-access transactions. In this case, neither the outside world nor any other individual user transaction can distinguish between the given system and the serial system. Note that the definition of serial correctness, relative to all non-access transactions, does not require that all transactions see schedules that are a part of the same execution of the serial system; rather each could see schedules arising in a different serial execution.

The serial correctness for all non-orphan transactions implies serial correctness for T_o because the serial scheduler does not have the action $ABORT(T_o)$ so T_o cannot be an orphan.

2.3 Transaction Automata

Each transaction is modeled as an I/O automaton with the help of the following operations [19]:

Input operations :	CREATE(T)
	REPORT-COMMIT(T' , v), where $T' \in \text{children}(T)$ and v is the return value
	REPORT-ABORT(T') where $T' \in \text{children}(T)$
Output operations:	REQUEST-CREATE(T') where $T' \in \text{children}(T)$
	REQUEST-COMMIT(T , v) where v is the return value

The CREATE operation wakes up the transaction. The REQUEST-CREATE is a request by T to create a particular child transaction. The REPORT-COMMIT operation reports to T the successful completion of one of its children and returns a value depending upon the operation performed. The REPORT-ABORT operation reports to T the unsuccessful completion of one of its children, without returning any value. The REQUEST-CREATE is a request by T to create a particular child transaction.

Generic Object Automata

The generic object automata [19] serve as the specifications of the concurrent behavior of the operations on the data objects. The operations for each object are the CREATE, REQUEST-COMMIT operations for all the corresponding access transactions. The CREATE operation is an invocation of an access to the object, while the RE-

QUEST-COMMIT is a return of value in response to such an invocation. It has two additional input operations INFORM-COMMIT and INFORM-ABORT for every transaction which inform about the fate of each transaction.

Input Operation: CREATE(T)
 INFORM-COMMIT-AT-(X) OF T, $T \neq T_0$
 INFORM-ABORT-AT-(X) OF T
Output Operation: REQUEST-COMMIT(T, v) where $T \in \text{accesses}(X)$

Generic Scheduler Automaton

The generic scheduler [19] is modeled as an I/O automaton. It passes requests for the creation of subtransactions to the appropriate recipient, makes decision about the completion of children and reports back to their parents, and informs objects of the fate of transactions. The operations are as follows:

Input operation : REQUEST-CREATE(T), $T \neq T_0$
 REQUEST-COMMIT(T, v)
Output operations : CREATE(T)
 COMMIT(T), $T \neq T_0$
 ABORT(T), $T \neq T_0$
 REPORT-COMMIT(T, v), $T \neq T_0$ and v a value
 REPORT-ABORT(T), $T \neq T_0$
 INFORM-COMMIT-AT-(X) OF T, $T \neq T_0$
 INFORM-ABORT-AT-(X) OF T, $T \neq T_0$

The REQUEST-CREATE and REQUEST-COMMIT inputs are identified with the corresponding output operations of transactions and object automata, and corresponding for the CREATE, REPORT-COMMIT and REPORT-ABORT output operations. The COMMIT and ABORT operations are internal, marking the point after which the decision on the fate of the transaction is irreversible. The COMMIT(T) and ABORT(T) are called return operations. The INFORM-COMMIT and INFORM-ABORT informs the data object automaton about the fate of transactions. For more details see [19].

3. NESTED TRANSACTION MODEL

Our nested transaction database system model formally consists of the transaction managers (TMs), recovery managers (RMs) and data managers (DMs). The data objects are modeled by the data managers (DMs). Each data manager keeps a copy of the data object in the secondary storage, called stable-db. The prewrite and write values of each object are kept in the respective buffers at the corresponding DMs. These are called prewrite- and write-buffers, respectively. Physically, only a subset of these DMs will have the prewrite and write values of the data objects in the corresponding buffers. A read operation gets the value of the referenced data object from the prewrite-buffer (if any), otherwise it gets the value from the write-buffer. If the DM does not have a copy of the data object in the write-buffer, the read operation gets the value from the stable-db copy of the data object. The write-buffer's contents of a data object are transferred periodically

to the stable storage. Also, each DM maintains a log corresponding to the data object. Each DM also shares a common log.

Based on the above configuration, our model has two different transaction managers for performing the read (read-TM) and write (write-TM) operations. These read- and write-TMs are initiated by the user-visible transactions. A hidden daemon transaction is associated with each write-TM transaction to co-ordinate the buffer management operations, i.e., the transfer of a data object’s value to the stable-db during the normal operations. To achieve the notion of spontaneity and transparency of buffer management operation, the daemon transaction wakes up and commits with respect to its associated transaction. Therefore, during the span of a daemon transaction, a daemon can initiate many transfer-RMs at the next level of transaction nesting.

TMs are situated at one level below user-visible transactions. The next level of transaction hierarchy has four different recovery managers for co-coordinating read (read-RM), prewrite (prewrite-RM), write (write-RM), transfer (transfer-RM) operations. These RMs initiate access subtransactions at the leaf level. Each read-, prewrite- and write-RM initiates read, prewrite and write access subtransactions, respectively. A read access reads the value either from the prewrite- or the write-buffer of the data object whereas a write subtransaction accesses only the write-buffer component of the data object. A transfer-RM initiates a transfer access (like read access, but returns no value) to transfer the contents of a write-buffer of the data object to the stable-db. The nested transaction tree structure is shown in Fig. 1.

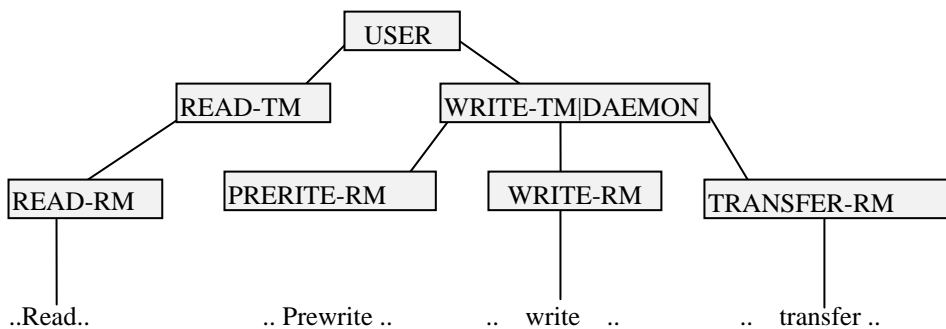


Fig. 1. Nested transaction tree.

We assume that each user-visible transaction knows its write-set before initiating a write-TM in order to write all the data objects in its write-set. Write-TM first initiates a prewrite-RM, which further initiates the prewrite access subtransactions in order to announce the prewrites for all the data objects contained in the write-set. This value for each data object is written in the prewrite-buffer allocated in the volatile memory. Modeling prewrites at leaf level provides user transparency to the prewrite operations.

We specify the prewrite-RM as the recovery point subtransaction of the top-level transaction. Once the prewrite-RM has committed, the prewrite values become visible outside its parent’s view. After the prewrite-RM’s commit, the write-TM initiates a write-RM to update all the data objects whose prewrite values have been announced be-

fore. The final updates are written in the write-buffers allocated in the volatile memory at each DM. With the invocation of a write-TM automaton, a daemon transaction is activated automatically which further initiates transfer-RMs. A transfer-RM initiates a transfer access subtransaction to transfer the write-buffer's value to the stable-db. The write-buffer's contents can be transferred without the commit of the top-level transaction because write-values, once written, cannot be undone or compensated.

4. CONCURRENCY CONTROL AND LOCKING

In this section, we mainly discuss the type of conflicts which occur in our model during normal operations and the locks needed to control them.

A read access transaction can read the value from the prewrite-buffer if it has the prewritten value of the corresponding data object. Otherwise, it gets the value from the write-buffer. A transfer access also reads a write-buffer's value to transfer it to the stable-db. The prewrite and write access subtransactions access prewrite- and write-buffers, respectively.

A prewrite operation introduces some more conflicting operations apart from the usual read-write and write-write conflicts. The three pairs of conflicting operations are prewrite-prewrite, prewrite-write and read-prewrite (only if read returns the prewrite-buffer's value). A prewrite-prewrite conflict occurs due to the fact that a prewrite value cannot be changed unless its associated write is performed. When a write operation is operating, its associated prewrite value cannot be changed or vice-versa. This is because a write operation replaces the value of the data object in the write-buffer by the prewrite-buffer's value. Hence, the prewrite and write operations cannot be executed concurrently. Also, while reading a data object's prewrite value, no other prewrite access transaction can change its prewrite value or vice-versa, otherwise, a read operation can get an inconsistent value. A transfer access is similar to a read access and hence, it does not introduce any new pair of conflicting operations.

To control concurrent read, prewrite, write and transfer accesses, each access subtransaction has to acquire its respective lock before accessing a data object. Our algorithm uses a read-lock for read and transfer access operations, and write- and prewrite-locks for write and prewrite access operations, respectively. Prewrite- and write-locks are exclusive locks, whereas a read-lock is a shared lock.

In our locking protocol, a transaction may hold and retain locks. A transaction holding a lock for an object is allowed access the corresponding object. The object is not allowed access if a transaction only retains the lock. Since the accesses are situated at leaf level only, all the holders of locks are at leaf level only. A retained lock is only a place-holder indicating that transactions outside the hierarchy of the retainer cannot acquire the same lock or any of its conflicting locks, but descendants of the retainer can acquire the same or non-conflicting locks.

Whenever a prewrite access transaction commits, it passes its prewrite-lock to its parent transaction (prewrite-RM). The prewrite-lock is passed to the parent so that the transactions outside the parent's hierarchy cannot get the prewrite-lock. This is because in case an upper level transaction of the committed prewrite access subtransaction aborts or system fails, the prewrite value is to be discarded. However, whenever the recovery

point subtransaction (prewrite-RM) commits, the committed subtransaction's lock is passed directly to the least common ancestor (of other waiting accesses and the committed transaction) without necessarily the commit of all its superior transactions up to the least common ancestor (l.c.a.). This also holds in case of commit of the recovery point subtransaction's superior transactions. The l.c.a. of waiting accesses and of committed transaction is determined dynamically at run time. The prewrite-lock cannot be released entirely because a new prewrite operation for the same data object cannot be initiated unless the write operation corresponding to the last prewrite is committed.

Locks inherited by a l.c.a. enables waiting read and write access subtransactions to acquire their respective locks early. This further increases the availability since a waiting access subtransaction T_2 can get its respective lock before all the ancestors of the committed transaction T_1 up to the least common ancestor of T_1 and T_2 are necessarily committed. For similar reasons as stated before, whenever a write access transaction commits, its lock is also passed to the least common ancestor of other waiting access subtransactions to enable them to acquire their respective locks. This also holds for all the ancestors of the committed write access subtransactions in case of their commit. Passing the lock early to l.c.a. is safe because once the recovery point subtransaction is committed, its higher-level transactions have to commit. It seems that once a transaction is committed, its write-lock can be released entirely, but this is not desirable for the following reasons. First, a transaction cannot get a write-lock unless its ancestors hold prewrite- and read-locks for the object. Therefore, releasing a write-lock does not help unless the other conflicting locks are released. Second, in case of restart, some transactions might need the write-locks to complete their interrupted execution due to failure.

The prewrite-lock held by a prewrite access subtransaction is released in case it, or any of its ancestors, aborts or the system fails because its effect is to be discarded. However, when a write access subtransaction aborts after it acquires the write-lock, its lock is passed to its parent. Furthermore, when a transaction aborts, the parent of the aborted transaction inherits the write-lock held by any of its descendants. In effect, the write- and/or prewrite-lock held by any transaction (except by prewrite access subtransactions) is not released entirely in case the lock retaining transaction aborts or system crashes. Releasing a prewrite- or a write-lock entirely in case of aborts or system crash may create an inconsistent state. This is due to the fact that some subtransactions might have to complete their remaining execution on revival. Read-only transactions can release their locks entirely in case they abort and pass their locks to parent transactions on commit. A transfer access releases its lock to the daemon transaction on commit, as its effects are not going to be discarded in case of aborts at higher-level or system crash. In the case of a transaction abort in the hierarchy of daemon transactions, its lock is released entirely. The daemon transaction releases its lock according to its associated transaction.

Formally, we have the following locking rules:

1. A read access can acquire a read-lock only if the prewrite-lock and write-lock on the corresponding DM is retained by its ancestor transaction.
2. A prewrite access subtransaction can get its respective prewrite-lock only if the prewrite-, write- and read-locks are retained by its ancestor transactions.
3. A write access transaction can get its write-lock only if read-, prewrite-, write-locks are held by its ancestors.

4. A transfer access can get a read-lock only if its ancestors hold the write-lock. Note that this rule is similar to rule 1.
5. When a read access subtransaction commits, it releases its lock to its parent. When its parent commits, it passes the lock to its parent, and so on.
6. When a transfer access transaction commits, it passes its lock to the daemon transaction. When a daemon commits, it passes its lock according to its associated transaction.
7. When a prewrite access commits, it passes its lock to its parent. When its parent commits (prewrite-RM), it passes the lock to the least common ancestor of all other access descendants waiting for the lock.
8. When a write access transaction commits, it releases its lock to the least common ancestor of all other access descendants waiting for the lock.
9. When a write access or prewrite-RM or any of its ancestor transaction aborts, the aborted transaction's locks are passed to its parent transaction.
10. When a read-only transaction aborts, its lock is released entirely. When a transaction in the hierarchy of daemon transaction aborts, its lock is released entirely.

4.1 Overview of Our Crash Recovery Algorithm

In this section, we give a brief outline of our recovery algorithm. We have used some recovery aspect while modeling different components as I/O automata. For example, we log various information during the processing of normal transactions as they are required for recovery. In real executions, they are logged during execution of transactions. We start with the requirements of our system crash recovery algorithm: revival of the database state of those data objects which do not contain their last committed values with respect to the execution up to the system failure.

- Revival of the prewrite values (kept in prewrite-buffers) of the data objects which have been announced by the committed recovery point subtransaction before system failure.
- In order to identify such data objects, the dirty object table has to be revived. The dirty object table is used to keep track of those data objects whose finally written values are inconsistent with the stable database values. This table also keeps information about those data objects whose prewrite values, announced by the committed recovery point subtransactions, have not been subsequently written to the database before a system crash.
- A system crash creates the additional problem of accomplishing the completion of those top-level transactions whose recovery point subtransactions have been committed before system crash. They have to reacquire the locks held by them at the time of failure before new transactions acquire such locks.
- To handle the above, the transaction and lock tables have to be revived. The transaction table keeps a list of all active transactions in the system at any time. The revived transaction table will recognize those active top-level transactions (and their active descendants) whose recovery point subtransactions have been committed before failures. The lock table contains the type of locks held by the transactions on different data objects at any time. The revived lock table will help in reacquiring the locks held by active top-level transactions and their descendants at the time of failure.

- To initiate new top-level transactions as soon as the dirty object, transaction, and lock tables and consistent states of prewrite- and write-buffers of dirty data objects are re-established.

The dirty object table is required to be checkpointed periodically by transferring a copy of it to the stable storage during normal processing. The prewrite values and after-images are logged on stable storage during the execution of transactions to build the consistent dirty object table in case a system failure occurs before the next checkpoint is taken. A transaction is not permitted to complete its commit processing until the redo portion of that transaction has been written to stable storage. The redo portion of a log record provides information on how to redo changes performed by the committed transactions. The prewrite values are logged on the commit of the recovery point subtransaction. During system restart, the dirty object table is recovered with the help of the most recent checkpointed copy of the dirty object table and is modified with the help of log stored after the last checkpoint.

The transaction and lock tables are checkpointed by transferring a copy of each of them to the stable storage periodically during normal processing. Whenever a subtransaction is made active or when any transaction acquires or releases a lock, the information is also logged to build a consistent state of these tables. However, such information may not be logged for read-only and prewrite access subtransactions as these transactions are to be discarded in case of a system failure. If a checkpoint is taken during restart recovery, then the contents of transaction and lock tables will also be included in the checkpoint. The entries corresponding to all other transactions except those that are to be restarted are removed from the transaction and lock tables. To do so, we need to find whether the stable storage contains the commit-state of the recovery point subtransaction of each active top-level transaction. The commit states information is transferred to the stable storage during the commit of those subtransactions whose effects cannot be undone or lost in case of a failure.

The commit-state information contains, besides associated variables, private data and other information, the identifier of the committed subtransaction as well as of its parent transaction. A commit-state information of a subtransaction T1 defines the state of its parent transaction T2 at the time of commit of T1. The commit-state information helps in re-establishing the restart state of a top-level transaction in order to complete its remaining execution. No subtransaction whose effects cannot be undone in case of failure can be considered complete until its commit-state information and all its data are safely recorded on stable storage. If the stable storage does not contain the commit-state of the recovery point subtransaction of an active top-level transaction, then all the entries corresponding to it and all its subtransactions are removed from the table. Otherwise, the top-level transaction has to complete its remaining execution upon revival.

To revive the contents of write-buffer of a dirty data object, we copy the value of the data object from the stable-db to the write-buffer. However, the stable database version of the data object may not contain some or all of the updates for committed transactions. It involves redoing those committed transactions after-images, which have not been transferred to the stable-db before the failure. The redo of after-images will re-establish the state of the database in the write-buffer at the time of failure. Similarly, to recover the prewrite-buffers corresponding to dirty data objects, we redo the prewrite

values logged on stable storage after the last checkpoint which have not subsequently been written. This re-establishes the states of prewrite-buffers of dirty data objects as they existed at the time of failure. The contents of prewrite- and write-buffers are recovered using the non-volatile storage version of the database, dirty object table and the log. There is at the most one prewrite log corresponding to a data object since once the associated write values are written, the corresponding prewrite log entry is removed from the stable storage as well as from the dirty object table.

To complete the active top-level transactions, whose recovery point subtransactions have been committed before system failure, the scheduler has to decide the restart states to reinitiate such transactions. If the recovery point subtransaction's commit is the only commit record in the stable log, then its active top-level transaction restarts from this commit-state. Otherwise, the scheduler finds the last commit-state logged after the commit of recovery point subtransaction prior to system crash in order to restart the transaction from the last commit-state. Once the restart-state is established, the scheduler reacquires the type of locks the active top-level transaction and all its active descendants were holding at the time of failure. Once the locks are reacquired, the execution of a top-level transaction restarts from the restart-state.

5. NORMAL SYSTEM OPERATIONS

The nested transaction model can be formally classified as a 4-tuple $(\tau, \text{parent}, O, V)$, where τ is a set of transaction names organized in a tree as shown in Fig. 1 by the mapping parent: $\tau \rightarrow \tau$. The mapping defines TMs as children of user-visible transactions, RMs as children of TMs, and access transactions as children of RMs. Each read-TM has read-RMs as its children, whereas each write-TM has prewrite-, write- and transfer-RMs as its children. Each read-RM, prewrite-RM, write-RM and transfer-RM has read, prewrite, write and transfer accesses as their children, respectively. O is the set of data objects and V is the set of returned prewrite or write values in our nested transaction system. In our model, all three types of components, namely non-access transactions, data managers, and the scheduler, are modelled as I/O automata.

Here we give the formal specifications for transaction automata involved in normal transaction processing. We first describe read-, write-TM and daemon automata at the level of TM, then give detailed specifications of read-, prewrite-, write-, and transfer-RM automata. Our automata also log the information required to deal with system crash such as transaction table, lock table, dirt object table, etc., during normal transaction processing. These are listed in the preconditions before the execution of an action and are updated after the execution of an operation in the postconditions.

5.1 Transaction Automata

Each TM and RM is modeled as a transaction automaton using the input and output operations given before.

The transition relation, denoted by (s', π, s) , has pre and post conditions given separately for each I/O operation π ; s' is the state before and s is the state after π . Any I/O operations having no pre and/or post conditions implies that all components in states s' and s are the same.

Each state of the automaton uses a subset of the components defined as follows:

1. active is a Boolean variable used to initiate a CREATE operation;
2. latch is a Boolean variable used to control a short term lock [MHLPS] held by a transaction on the transaction, lock and dirty data object tables;
3. read is a Boolean variable used to control the initiation of read-TMs and commit of read-RMs;
4. prewritten is a Boolean variable used to control the initiation of write-RMs;
5. written is a Boolean variable used to control the REQUEST-COMMIT operation of write-RM.

Initially, all of the above Boolean variables are set to false.

6. read-req, write-req, prewrite-req and transfer-req are the sets of requested read-, write-, prewrite- and transfer-RMs, respectively;
7. read-act, write-act, prewrite-act and transfer-act are the sets of requested read, write, prewrite and transfer accesses, respectively;
8. pre-commit is the set of committed prewrite accesses;
9. written-1 is the set of updated data objects;
10. read-1 keeps the values of data object read.

Initially, all of the above sets are empty.

11. data is the value component of the data object returned in response to a read operation. Initially, data is undefined;
12. prewrite-set and write-set are the sets of data objects to be prewritten and written, respectively;
13. pre-value is the value of the data object to be prewritten;
14. TT is the transaction table in the form of an array shared by all DMs;
15. CLog is the common log shared by all DMs.

Initially, TT and CLog are as they exist before the operation π .

Read-TM

A read-TM performs the logical read accesses on behalf of the user-visible transactions. It returns the value read to the user-visible transaction.

Each state has active, latch, read-req, read, TT and data as components used to define pre and post conditions for any I/O operation. The I/O operations are defined as follows:

o CREATE(T)

To initiate a CREATE(T) operation, the state s' in the first precondition has the variable active set to false, indicating that T has not been created before. The second precondition checks that no checkpoint operation is in progress, i.e., the latch associated with the transaction table must have a value false. When T is activated by a CREATE(T) operation with the given preconditions satisfied, the state of the automaton changes from s' to s . active which then becomes true. The next postcondition says that an entry of the type $\langle T, \text{active} \rangle$ is made in the transaction table. The transaction table keeps information about active transactions in the system, which helps in the recovery process in case of system failure.

Precondition: $s.active = false$
 $latch(TT) = false$
 Postcondition: $s.active = true$
 $TT = TT \cup \langle T, active \rangle$

o REQUEST-CREATE(T') where T' is a read-RM

The first precondition ensures that CREATE(T) has already occurred. The second condition checks that T' should not have already been request-created, as the same transaction cannot be initiated again. Given the above preconditions, REQUEST-CREATE operation for T' is initiated and this information is recorded by updating the variable read-req in the postcondition.

Precondition: $s.active = true$
 $T' \in s.read-req$
 Postcondition: $s.read-req = s.read-req \cup \{T'\}$

o REPORT-COMMIT(T', v) where T' is a read-RM

This has no preconditions. In the postconditions, the variable read is set to true indicating that the read process is over and the variable data has the value read.

Postcondition: $s.read = true$
 $s.data = v$

o REPORT-ABORT(T') where T' is a read-RM

This has no postcondition as it is not necessary for correctness to remember the aborted read-RMs.

Postcondition: no change

o REQUEST-COMMIT(T, v)

In the precondition, active and read become true, and v is the value returned. In the postcondition, active is set to false indicating that read-TM has committed.

Precondition: $s.active = true$
 $s.read = true$
 $v = s.data$
 Postcondition: $s.active = false$

Write-TM

A write-TM performs the logical write accesses on behalf of user-visible transactions. It coordinates the announcement of prewrite values for each data object before finally updating the data objects.

A write-M automaton has active, latch, prewritten, written, prewrite-req, write-req, prewrite-set, write-set, TT, CLog as state components used in defining pre and post conditions for the following I/O operations:

o CREATE(T)

The preconditions are as given in the case of CREATE operation for a read-RM. The first two postconditions are as given earlier. The last postcondition asserts that T becomes active and is placed in the common log.

Precondition: $s'.active = false$
 $latch(TT) = false$
 Postcondition: $s'.active = true$
 $TT = TT \cup \{<T, active>\}$
 $CLog = CLog \cup \{<T, active>\}$

o REQUEST-CREATE(T') where T' is a prewrite-RM and prewrite-set(T') = q

The first precondition is as before. The second precondition says that REQUEST-CREATE for T' should not have previously occurred. Given the above preconditions, REQUEST-CREATE for T' is initiated, and this information is recorded by updating the variable prewrite-req in the postcondition.

Precondition: $s'.active = true$
 $T' \in s'.prewrite-req$
 Postcondition: $s'.prewrite-req = s'.prewrite-req \cup \{T'\}$

o REPORT-COMMIT(T', v) where T' is a prewrite-RM

The postcondition asserts that T' has been committed.

Postcondition: $s'.prewritten = true$

o REPORT-ABORT(T') where T' is a prewrite-RM

This has no postcondition as it is not necessary for correctness to remember the aborted prewrite-RMs.

Postcondition: no change

o REQUEST-CREATE(T') where T' is a write-RM and write-set(T') = d

The pre and post conditions are as already explained.

Precondition: $s'.active = true$
 $s'.prewritten = true$
 $T' \in s'.write-req$
 Postcondition: $s'.write-req = s'.write-req \cup \{T'\}$

o REPORT-COMMIT(T', v) where T' is a write-RM

Postcondition: $s'.written = true$

o REPORT-ABORT(T') where T' is a write-RM

This I/O operation is very important in our algorithm. Note that REPORT-ABORT

of a prewrite-RM has no preconditions. This is not true in the case of an abort of a write-RM. The scheduler can accept a REPORT-ABORT for a write-RM T' only if write access subtransaction T'' of T' has committed, which precisely is the precondition.

Precondition: No write access $T'' \in \text{children}(T')$ has committed

Postcondition: no change

o REQUEST-COMMIT(T, v)

This I/O operation returns the value $v = \text{nil}$ to the user-visible transaction.

Precondition: $s.\text{active} = \text{true}$

$v = \text{nil}$

$s.\text{written} = \text{true}$

Postcondition: $s.\text{active} = \text{false}$

Daemon Automaton

A daemon automaton is associated with each write-TM automaton. A daemon automaton is not a transaction automaton but is a part of a write-TM automaton. It gets activated with its associated TM automaton, say T , and goes to sleep when T makes a request to commit. That is, a daemon automaton does not issue a commit, but instead, it receives the REQUEST-COMMIT as an input operation. A daemon transaction invokes a transfer-RM which perform the transfer of a data object's value from the write-buffer to the stable-db. The associated TM has no control over the transfer of data objects in the sense that it cannot observe whether or not the transfer of a data object's write-buffer value to the stable-db has been successful.

Each state has active, transfer-act, latch as components used to define pre and post conditions for any I/O operation. The I/O operations are defined as follows:

Input Operations: CREATE(T)

REPORT-COMMIT(T', v), where $T' \in \text{children}(T)$ and v is the return value

REPORT-ABORT(T') where $T' \in \text{children}(T)$

REQUEST-COMMIT(T, v) where v is the return

Output Operations: REQUEST-CREATE(T') where $T' \in \text{children}(T)$

o CREATE(T)

Precondition: $s.\text{active} = \text{false}$

$\text{latch}(TT) = \text{false}$

Postcondition: $s.\text{active} = \text{true}$

$TT = TT \cup \{<T, \text{active}>\}$

o REQUEST-CREATE(T') where T' is a transfer-RM

Precondition: $s.\text{active} = \text{true}$

$T' \in s.\text{transfer-act}$

Postcondition: $s.\text{transfer-act} = s.\text{transfer-act} \cup \{T'\}$

o REPORT-COMMIT(T', v)

Postcondition: no change

o REPORT-ABORT(T')

Postcondition: no change

o REQUEST-COMMIT(T, v)

Postcondition: s.active = false

Read-RM

A read-RM performs logical read accesses on the DMs, and returns the value read to the corresponding read-TM.

Each state has components active, latch, read-act, TT, read-1 and data used to define pre and post conditions. The I/O operations are defined as follows:

o CREATE(T)

Precondition: s'.active = false
 latch(TT) = false
 Postcondition: s.active = true
 $TT = TT \cup \{<T, \text{active}>\}$

o REQUEST-CREATE(T') where T' is a read access

Precondition: s'.active = true
 $T' \in s'.\text{read-act}$
 Postcondition: $s.\text{read-act} = s'.\text{read-act} \cup \{T'\}$

o REPORT-COMMIT(T', v)

This operation returns the value v to its parent transaction read-TM. In the postcondition, the variable s.read = o(T') (refers to the object accessed by T') gives the object read by the transaction T' and returns the value $v = s.\text{data}$.

Postcondition: s.read-1 = o(T')
 s.data = v

o REPORT-ABORT(T')

Postcondition: no change

o REQUEST-COMMIT(T, v)

Precondition: s'.active = true
 s'.read-1 = o(T')
 $v = s'.\text{data}$
 Postcondition: s.active = false

Prewrite-RM

A prewrite-RM announces the prewrite values for all data objects as instructed by its parent transaction write-TM.

Components in each state of prewrite-RM automaton are active, latch, pre-act, pre-value, pre-commit and TT. Each data object to be prewritten belongs to q , the prewrite-set of data objects. The I/O operations are defined as follows:

o CREATE(T)

Precondition: $s'.active = false$
 $latch(TT) = false$
 Postcondition: $s'.active = true$
 $TT = TT \cup \{<T, active>\}$
 $CLog = CLog \cup \{<T, active>\}$

o REQUEST-CREATE(T') where T' is a prewrite access and pre-value(T') = $p_x \in q$

Precondition: $s'.active = true$
 $T' \in s'.pre-act$
 $pre-value(T') = p_x$
 Postcondition: $s'.pre-act = s'.pre-act \cup \{T'\}$

o REPORT-COMMIT(T', v)

Postcondition: $s'.pre-commit = s'.pre-commit \cup \{o(T')\}$

o REPORT-ABORT(T')

Postcondition: no change

o REQUEST-COMMIT(T, v)

Precondition: $s'.active = true$
 $q = s'.pre-commit$
 $v = nil$
 Postcondition: $s'.active = false$

Write-RM

A write-RM initiates the write accesses on data objects whose prewrite values have been announced by the prewrite-RM and returns a value nil to the corresponding write-TM.

Each state has components active, latch, write-act, written-1, TT and CLog in pre and post conditions for the following I/O operations:

o CREATE(T)

Precondition: $s'.active = false$
 $latch(TT) = false$
 Postcondition: $s'.active = true$
 $TT = TT \cup \{<T, active>\}$
 $CLog = CLog \cup \{<T, active>\}$

o REQUEST-CREATE(T') where T' is a write access

Precondition: $s'.active = true$
 $T' \in s'.write-act$
 Postcondition: $s.write-act = s'.write-act \cup \{T'\}$

o REPORT-COMMIT(T', v)

Postcondition: $s.written-1 = s'.written-1 \cup \{o(T')\}$

o REPORT-ABORT(T')

Postcondition: no change

o REQUEST-COMMIT(T, v)

Precondition: $s'.active = true$
 $s'.written-1 \in write-set(T)$
 $v = nil$
 Postcondition: $s.active = false$

Transfer-RM

A transfer-RM is invoked by the daemon automaton which copies the current value of the data object from the write-buffer to the stable-db and returns no value to the parent transaction. When a transfer access commits, the transfer-RM can announce its own commit.

Each state has active, latch, TT and transfer-act as components used to define pre and post conditions for any I/O operation. The I/O operations are defined as follows:

o CREATE(T)

Precondition: $s'.active = false$
 $latch(TT) = false$
 Postcondition: $s.active = true$
 $TT = TT \cup \{<T, active>\}$

o REQUEST-CREATE(T') where T' is a transfer access

Precondition: $s'.active = true$
 $T' \in s'.transfer-act$
 Postcondition: $s.transfer-act = s'.transfer-act \cup \{T'\}$

o REPORT-COMMIT(T', v)

Postcondition: no change

o REPORT-ABORT(T')

Postcondition: no change

o **REQUEST-COMMIT(T, v)**

Precondition: $s.active = true$

$v = nil$

Postcondition: $s.active = false$

5.2 DM Automaton: Normal System Operations

Recall that each DM automaton has a copy of the data object in the write-buffer as well as on stable storage, called stable-db. Each DM automaton also maintains a prewrite-buffer which keeps the prewrite value of the corresponding data object. Each DM automaton controls concurrent accesses using the locking rules described earlier. All DMs share lock, transaction, dirty object tables and the CLog information.

As our data objects handle concurrent operations, they are modelled by generic object automata. Input and Output operations are as given for a generic object automaton. Here, there are two more operations, an input action for initiating a checkpoint operation and an output action for announcing the end of the checkpoint activity. These actions are as follows:

Input Operation: BGN-CHK-POINT(T') where T' is a checkpoint transaction

Output Operation: END-CHK-POINT(T') where T' is a checkpoint transaction

Each state has active, chk, read-run, prewrite-run, write-run, prewrite-buffer, write-buffer, Log_x , CLog, read-lockholders, write-lockholders, prewrite-lockholders, LT, DOT, TT, stableTT, stableDOT, stableLT, logLSN, stableLSN, stable-db, latch as components used to define pre and post conditions for any I/O operation. chk is a Boolean variable to control the checkpoint operations read-run is a set of request-committed read access subtransactions, prewrite-run (write-run) is a set of request committed prewrite (write) access subtransactions. LT is the lock table in the form of an array shared by all DMs, logLSN is the log sequence number stored with each log record in Log_x (Log for the data object X), stableLSN is the LSN of the write-buffer at the time stable-db value of the data object is updated. The CLog and Log_x denote common log and the log corresponding to each data object, respectively. The lock, transaction and dirty object tables are also modelled as arrays. CLog, Log_x , stableLSN, stable-db, stableDOT, stableTT, stableLT are the stable storage variables (i.e., their values are kept on stable storage). Initially, logLSN = 0, stableLSN = 0, and all the Boolean variables are set to false, whereas TT and its stable copy are as before the CREATE(T) operation. Initially, LT, DOT and their stable copies, and Log_x are empty. All other sets are empty.

Some of the following actions contain preconditions in which the function least is applied to the set s.write-lockholders. In case least() is undefined, the precondition is assumed to be false. Also, l.c.a.(T, accesses) denotes the least common ancestor of the transaction T and all the access subtransactions. A transaction belonging to set write-lockholders holds a write-lock. Similarly, a transaction in read-lockholders (prewrite-lockholders) holds a read-lock (prewrite-lock). The I/O operations are defined as follows:

o CREATE(T)

Precondition: $\text{latch}(TT) = \text{false}$
 Postcondition: $s.\text{active} = s'.\text{active} \cup \{T\}$
 $TT = TT \cup \{<T, \text{active}>\}$

o REQUEST-COMMIT(T, v) where T is a read access

The first precondition says that latch value associated with the lock table must be false. That is, no checkpoint operation must be in progress. The second precondition asserts that the read access transaction must be active but not completed. The next precondition checks that the conflicting write- and prewrite-locks must be with the ancestors of T according to the locking rules. The next condition checks whether or not the data object being accessed has associated prewritten value in the state s' announced by $\text{least}(s'.\text{prewrite-lockholders})$. The non-null value of the prewrite-buffer in the state s' indicates that the prewrite value has not been written in the object's write-buffer. Therefore, the read access must read the value from the prewrite-buffer. Otherwise, it reads the value from the object's write-buffer. If write-buffer = null, i.e., if DM has no copy of the object in the write-buffer, then it returns the value from the stable-db copy of the data object. When these preconditions are satisfied, the automaton changes its state from s' to s with the following postconditions. The first asserts that REQUEST-COMMIT for the transaction T is initiated, and this information is recorded by adding T to the set read-run. In the next precondition, the set read-lockholders is updated to capture the fact that a read-lock is granted to T. The next postcondition says that an entry is made in the lock table.

Precondition: $\text{latch}(LT) = \text{false}$
 $T \in s'.\text{active} - s'.\text{read-run}$
 $s'.\text{write-lockholders} \cup s'.\text{prewrite-lockholders} \subseteq \text{ancestors}(T)$
 If $s'.\text{prewrite-buffer}(\text{least}(s'.\text{prewrite-lockholders})) \neq \text{null}$, then
 $v = s'.\text{prewrite-buffer}(\text{least}(s'.\text{prewrite-lockholders}))$
 If $s'.\text{prewrite-buffer}(\text{least}(s'.\text{prewrite-lockholders})) = \text{null}$, then
 $v = s'.\text{write-buffer}(\text{least}(s'.\text{write-lockholders}))$
 If $s'.\text{write-buffer}(\text{least}(s'.\text{write-lockholders})) = \text{null}$, then
 $v = \text{stable-db}(o(T))$
 Postcondition: $s.\text{read-run} = s'.\text{read-run} \cup \{T\}$
 $s.\text{read-lockholders} = s'.\text{read-lockholders} \cup \{T\}$
 $LT = LT \cup \{<T, o(T), \text{read-lock}>\}$

o REQUEST-COMMIT(T, v) where T is a prewrite access

The first precondition says that CREATE(T) has occurred before and the transaction T has not been completed. The second condition ensures that the prewrite-, write- and read-locks must be with the ancestors of T according to the locking rules. Next, prewrite-buffer's null value indicates that write-buffer has been replaced by the prewrite-buffer's value. The last precondition asserts that the latch associated with the lock table must be false to ensure that no checkpoint operation is in progress. The first postcondition asserts that the new prewrite value is placed in the prewrite-buffer. The LSN value of the prewrite-buffer is also updated. The variable prewrite-run in state s

records that T is request-committed and prewrite-lockholders keep the record that a prewrite-lock is granted to T. The last postcondition says that the lock table is updated.

Precondition: $T \in s'.active - s'.prewrite-run$
 $s'.prewrite-lockholders \cup s'.write-lockholders \cup s'.read-lockholders$
 $\subseteq ancestors(T)$
 $s'.prewrite-buffer(least(s'.prewrite-lockholders)) = null$
 $latch(LT) = false$

Postcondition: $s'.prewrite-buffer(T) = pre-value(T)$
 $s'.prewrite-buffer(T).LSN = logLSN + 1$
 $s'.prewrite-run = s'.prewrite-run \cup \{T\}$
 $s'.prewrite-lockholders = s'.prewrite-lockholders \cup \{T\}$
 $LT = LT \cup \{<T, o(T), prewrite-lock>\}$

o REQUEST-COMMIT(T, v) where T is a write access

The first precondition is as before. The second precondition asserts that prewrite-, write- and read-locks must be with the ancestors of T according to the locking rules. Next, in state s', the prewrite-buffer's non-null value indicates that write access, so far, has not replaced the object's write-buffer value with the prewrite value of the object. In the next two preconditions, the value to be written is transferred to the log and the write-buffer along with LSN is updated. The latches associated with LT and DOT must be set to the value false to ensure that no checkpoint operation is active. The first postcondition asserts that the prewrite-buffer's value is set to null to indicate that the prewritten value has been transferred to the object's write-buffer. In the next two conditions, a write-lock is granted to T and this information is also recorded in the CLog in order to handle system failure. The last two postconditions say that the dirty data object and lock tables are updated.

Precondition: $T \in s'.active - s'.write-run$
 $s'.prewrite-lockholders \cup s'.write-lockholders \cup s'.read-lockholders$
 $\subseteq ancestors(T)$
 $s'.prewrite-buffer(least(s'.prewrite-lockholders)) \neq null$
 $s'.write-buffer(T) = s'.prewrite-buffer(least(s'.prewrite-lockholders))$
 $s'.write-buffer.LSN = logLSN + 1$
 $Log_{o(T)} = Log_{o(T)} \cup \{<LSN, T, o(T), s'.write-buffer(T)>\}$
 $latch(LT) = false$
 $latch(DOT) = false$

Postcondition: $s'.prewrite-buffer(least(s'.prewrite-lockholders)) = null$
 $s'.write-lockholders = s'.write-lockholders \cup \{T\}$
 $CLog = CLog \cup \{<T, o(T), write-lock>\}$
 $s'.write-run = s'.write-run \cup \{T\}$
 $latch(LT) = true$
 $latch(DOT) = true$
 $DOT = DOT - \{<o(T), LSN (prewrite), \perp>\} \cup$
 $\{<o(T), \perp, LSN (write)>\}$
 $LT = LT \cup \{<T, o(T), write-lock>\}$

o REQUEST-COMMIT(T, v) where T is a transfer access

The preconditions are as before. In the postconditions, the write-buffer's value of the object is placed in the stable-db along with LSN if the stableLSN associated with the write-buffer of the object is greater than the LSN of the write-buffer. The stableLSN of the write-buffer is also updated. The transaction T acquires a read-lock. The lock and dirty object tables are also updated. The last postcondition says that transfer-run is updated, which informs that T has been completed.

Precondition: $T \in s'.active - s'.transfer-run$
 $s'.write-lockholders \subseteq ancestors(T)$
 $latch(LT) = false$
 $latch(DOT) = false$

Postcondition: If $s.write-buffer.stableLSN, < s.write-buffer.LSN$ then
 $stable-db(o(T)) = s.write-buffer$
 $stable-db(o(T)).LSN = s.write-buffer.LSN$
 $s.write-buffer.stableLSN = s.write-buffer.LSN$
 $s.read-lockholders = s'.read-lockholders \cup \{T\}$
 $LT = LT \cup \{<T, o(T), read-lock>\}$
 $DOT = DOT - \{<log records> \text{ whose LSNs are less than or equal to } s.write-buffer.LSN\}$
 $s.transfer-run = s'.transfer-run \cup \{T\}$

o INFORM-COMMIT-AT-(X) OF T

The preconditions simply check that no checkpoint operation is currently in progress. Below, we have four sets of postconditions. Only one set of postconditions is, however, satisfied at a given time. Most of the postconditions deal with logging of information on stable storage.

The postcondition, common to all the sets of postconditions, says that if T is committed, its entry is removed from the transaction table.

The first postcondition in A says that if T belongs to the set of read-lockholders, then the read-lock held by the transaction T is released to the parent of T according to the locking rules. Next, the lock table is updated accordingly. Next, if T is a transfer access, then the lock is released to the daemon according to the locking rules.

Precondition: $latch(TT) = false$
 $latch(LT) = false$
 $latch(DOT) = false$

Postcondition: If $T \in TT$ then $TT = TT - \{<T, active>\}$

A. If $T \in s'.read-lockholders$ and T is not a transfer access, then
 $\{s.read-lockholders = s'.read-lockholders - \{T\} \cup \{parent(T)\}$
 $LT = LT - \{<(T), o(T), read-lock>\} \cup \{<parent(T), o(T), read-lock>\}$
 If $T \in s'.read-lockholders$ and T is a transfer access, then
 $\{s.read-lockholders = s'.read-lockholders - \{T\} \cup \{daemon\}$
 $DOT = DOT - \{<o(T), \perp, LSN>\}$

The postconditions in B are as follows: The first condition says that if T belongs to the set of all write-lockholders, then the write-lock held by transaction T is released to the l.c.a.(T, accesses) according to the locking rules and the relevant information is also placed in CLog and LT. Next, the write-buffer's value is passed to the l.c.a.(T, accesses). The last postcondition asserts that the transaction's commit information is also placed in CLog (if it is not already there), which helps in the recovery process during system crash.

B. If $T \in s'.write\text{-}lockholders$ then

$$\begin{aligned} & \{s.write\text{-}lockholders = s'.write\text{-}lockholders - \{T\} \cup \{(l.c.a(T, accesses))\}\} \\ & CLog = CLog \cup \{<l.c.a(T, accesses), o(T), write\text{-}lock>\} \\ & LT = LT - \{<T, o(T), write\text{-}lock>\} \cup \{<l.c.a(T, accesses), o(T), write\text{-}lock>\} \\ & s.write\text{-}buffer(l.c.a(T, accesses)) = s'.write\text{-}buffer(T) \\ & s.commit(parent(T) = s'.commit(parent(T)) \cup \{<T, commit>\}) \\ & \text{If } <T, commit> \in CLog \text{ then } CLog = CLog \cup \{<T, commit>\} \end{aligned}$$

In C, the first condition says that if the non-access transaction T belongs to the set of all prewrite-lockholders, then the prewrite-lock held by the transaction T is released to the l.c.a.(T, accesses) according to the locking rules, and this information is also stored in CLog and LT. Next, the prewrite-buffer's value is passed to l.c.a.(T, accesses). In the next condition, if T is a prewrite-RM then the prewrite value corresponding to the DM is also logged. The last condition asserts that if the commit information is not there in the log then it is recorded in the CLog.

C. If T is a non-access transaction and $T \in s'.prewrite\text{-}lockholders$, then

$$\begin{aligned} & \{s.prewrite\text{-}lockholders = s'.prewrite\text{-}lockholders - \{T\} \cup \{(l.c.a(T, accesses))\}\} \\ & CLog = CLog \cup \{<l.c.a(T, accesses), o(T), prewrite\text{-}lock>\} \\ & LT = LT - \{<T, o(T), prewrite\text{-}value>\} \cup \{<l.c.a(T, accesses), o(T), \\ & \quad prewrite\text{-}lock>\} \\ & \text{If } s.prewrite\text{-}buffer \neq \text{null} \text{ then} \\ & \{s.prewrite\text{-}buffer(l.c.a(T, accesses)) = s'.prewrite\text{-}buffer(T)\} \\ & \text{If } T \text{ is a prewrite-RM then} \\ & \{Log_x = Log_x \cup \{<LSN, T_i, o(T), Prewrite\text{-}value>\}\} \\ & s.commit(parent(T) = s'.commit(parent(T)) \cup \text{commit}(T)) \\ & \text{If } <T, commit> \in CLog \text{ then } CLog = CLog \cup \{<T, commit>\} \end{aligned}$$

Note that T_i is the committed prewrite access subtransaction which accessed this DM.

In D, the prewrite-lock is passed to the parent according to the locking rules and the lock table is updated. In the following two conditions, the prewrite-buffer's value is also passed to its parent and the commit information is also passed to its parent.

D. If T is an access transaction and $T \in s'.prewrite\text{-}lockholders$, then

$$\begin{aligned} & \{s.prewrite\text{-}lockholders = s'.prewrite\text{-}lockholders - \{T\} \cup \{parent(T)\}\} \\ & LT = LT \cup \{<parent(T), o(T), prewrite\text{-}lock>\} \\ & s.prewrite\text{-}buffer(parent(T)) = s.prewrite\text{-}buffer(T) \\ & s.commit(parent(T) = s'.commit(parent(T)) \cup \{<T, commit>\}) \end{aligned}$$

o INFORM-ABORT-AT-(X) OF T

The preconditions simply check that no checkpoint operation is currently in progress. The first postcondition asserts that an entry of the aborted transaction has been removed from the transaction table.

The postconditions in A say that if a read-lock is held by T and the descendants of T hold no write- or prewrite-locks, then all the descendants of T are discarded according to the locking rules and the lock table is also updated accordingly. Otherwise, the read-lock is passed to its parent and the lock table is also updated.

Precondition: $\text{latch}(TT) = \text{false}$
 $\text{latch}(LT) = \text{false}$
 $\text{latch}(\text{DOT}) = \text{false}$
 Postcondition: If $T \in TT$ then $TT = TT - \{ \langle T, \text{active} \rangle \}$

A. If $T \in s'.\text{read-lockholders}$ and $\text{descendants}(T) \in s'.\text{write-lockholders} \cup s'.\text{prewrite-lockholders}$, then
 $\{s.\text{read-lockholders} = s'.\text{read-lockholders} - \text{descendants}(T)$
 $LT = LT - \{ \langle T', o(T'), \text{lock-type} \rangle \text{ for all } T' \in \text{descendants}(T) \}$ else
 $\{s.\text{read-lockholders} = s'.\text{read-lockholders} - T \cup \{ \text{parent}(T) \}$
 $LT = LT - \{ \langle T, o(T), \text{read-lock} \rangle \} \cup \{ \langle \text{parent}(T), o(T), \text{read-lock} \rangle \}$

In B, if T holds a write-lock then the lock is released to its parent according to the locking rules and the lock table and CLog are updated.

B. If $T \in s'.\text{write-lockholders}$ then
 $\{s.\text{write-lockholders} = s'.\text{write-lockholders} - \{T\} \cup \{ \text{parent}(T) \}$
 $\text{CLog} = \text{CLog} \cup \{ \langle \text{parent}(T), o(T), \text{write-lock} \rangle \}$
 $LT = LT - \{ \langle T, o(T), \text{write-lock} \rangle \} \cup \{ \langle \text{parent}(T), o(T), \text{write-lock} \rangle \}$
 The postconditions in C are similar to B except that the lock here is a prewrite-lock.

C. If $T \in s'.\text{prewrite-lockholders}$ and T is an ancestor of prewrite-RM then
 $\{s.\text{prewrite-lockholders} = s'.\text{prewrite-lockholders} - \{T\} \cup \{ \text{parent}(T) \}$
 $LT = LT - \{ \langle T, o(T), \text{prewrite-lock} \rangle \} \cup \{ \langle \text{parent}(T), o(T), \text{prewrite-lock} \rangle \}$
 $\text{CLog} = \text{CLog} \cup \{ \langle \text{parent}(T), o(T), \text{prewrite-lock} \rangle \}$

In D, if the aborted transaction is a prewrite access or a prewrite-RM, then the prewrite-buffer's value is set to null. In the next postcondition, the lock table is updated.

D. If T is a prewrite access or a prewrite-RM and $T \in s'.\text{prewrite-lockholders}$, then
 $\{s.\text{prewrite-lockholders} = s'.\text{prewrite-lockholders} - \{T\}$
 $s.\text{prewrite-buffer}(T) = \text{null}$
 $LT = LT \cup \{ \langle T, o(T), \text{prewrite-lock} \rangle \}$

The following operations perform periodic checkpointing with the help of a checkpoint transaction T' initiated by the scheduler.

o BGN-CHK-POINT(T')

The first three preconditions check that latches are not acquired by any other checkpoint transaction on lock, transaction and dirty data object tables. No write access T is holding a write-lock on the DM, nor should a transfer access T hold a read-lock on the DM. In the postconditions, latches are acquired on the lock, transaction and dirty object tables and are updated. Also, the write-buffer's value is assigned to null, and $chk = true$ informs that checkpoint operation has been initiated.

Precondition: $latch(LT) = false$
 $latch(DOT) = false$
 $latch(TT) = false$
 $s.chk = false$
 T'' is a write access and $T'' \in s'.write-lockholders$
 T' is a transfer access and $T' \in s'.read-lockholders$

Postcondition: $latch(TT) = true$
 $latch(LT) = true$
 $latch(DOT) = true$
 $stableTT = stableTT \cup \{TT - stableTT\}$
 $stableLT = stableLT \cup \{LT - stableLT\}$
 $stable-db(o(T')) = s.write-buffer$
 $stable-db(o(T')).LSN = s.write-buffer.LSN$
 $DOT = DOT - \{\langle log\ record \rangle\ of\ the\ type\ "data"\ whose\ LSNs\ are\ less\ than\ or\ equal\ to\ s.write-buffer.LSN\}$
 $stableDOT = stableDOT \cup \{DOT - stableDOT\}$
 $s.write-buffer = null$
 $s.chk = true$

o END-CHK-POINT(T')

Precondition: $s'.chk = true$
 $CLog = Clog \cup \{\langle END-CHK-POINT \rangle\}$

Postcondition: $latch(LT) = false$
 $latch(DOT) = false$
 $latch(LT) = false$
 $s.chk = false$

6. BASIC PROPERTIES

We begin with some definitions which use the properties stated and proven below. These properties provide information about the states of transactions deducible from the data manager automaton of a data object X. These properties establish that the DM automaton correctly describes the concurrency control algorithm's behavior at the object interface. Many of these definitions and some of the properties follow the line of reasoning given in [2] and [18].

Let γ be a schedule of the DM for the data object X. Now we state the following result (without proof).

Lemma 1: Let γ be a schedule of the DM for the data object X , then γ is well-formed.

Definition 1: If γ is a well-formed sequence of operations of the DM for the data object X , T is a write or transfer access to X and T' is an ancestor of T , then we say that T is committed at X to T' in γ if γ contains INFORM-COMMIT-AT- (X) of T and the l.c.a. (T, T') has acquired the lock from T . Similarly, if T is a prewrite access to X and T' is an ancestor of T , then we say that T is committed at X to T' in γ if γ contains INFORM-COMMIT-AT- (X) of U where U is a parent of T and the l.c.a. (T, T') has acquired the lock from T .

If T is a read access to X and T' is an ancestor of T , we say that T is committed at X to T' in γ if γ contains a subsequence γ' consisting of an INFORM-COMMIT-AT- (X) of U event for every U that is an ancestor of T and a proper descendant of T' , arranged in ascending order so that the INFORM-COMMIT for parent(U) is preceded by that for U .

Definition 2: If γ is a well-formed sequence of operations of a DM and if T is an access to X , and T' is any transaction, we then say that T is visible at X to T' in γ if T is committed at X to l.c.a. (T, T') .

We denote by $\text{visible}_X(\gamma, T)$ the subsequence of γ that consists of operations of X whose transactions are visible at X to T in γ . Also, since γ is well-formed, $\text{visible}_X(\gamma, T)$ is a well-formed sequence of operations of the object X .

Definition 3: A transaction T is termed an orphan at X in γ if INFORM-ABORT-AT- (X) of U occurs in γ for some ancestor U of T .

Definition 4: Given any sequence of operations γ of X , we define $\text{write}(\gamma)$ to be the subsequence consisting of the REQUEST-COMMIT(T, v) for all prewrite and write accesses T .

Definition 5: Given any well-formed sequence γ of operations of the DM for the data object X , let $\text{essence}(\gamma)$ denote the sequence obtained from $\text{write}(\gamma)$ by placing a CREATE(U) event immediately preceding a REQUEST-COMMIT(U, v) event. Since γ is well-formed, $\text{essence}(\gamma)$ consists of a subset of events of γ and is well-formed. Clearly, γ and $\text{essence}(\gamma)$ are write-equal, i.e., $\text{write}(\gamma) = \text{write}(\text{essence}(\gamma))$.

The following fundamental property of the state of the DM expresses the fact that conflicting locks are never held by transactions except when one transaction is an ancestor of the other. This condition is enforced when locks are granted, and preserved thereafter by all actions.

Property 1: Let γ be a well-formed schedule of the DM for the data object X . Then the following hold:

- (a) Suppose $T \in s.\text{write-lockholders}$ and $T' \in s.\text{read-lockholders} \cup s.\text{write-lockholders} \cup s.\text{prewrite-lockholders}$, then either T is an ancestor of T' or else T' is an ancestor of T .

- (b) $T \in s.\text{prewrite-lockholders}$ and $T' \in s.\text{read-lockholders} \cup s.\text{write-lockholders} \cup s.\text{prewrite-lockholders}$, then either T is an ancestor of T' or else T' is an ancestor of T .

Proof of 1(a): A write-lock conflicts with other write-, read- and prewrite-locks. Therefore, if T is a holder (retainer) of a write-lock ($T \in s.\text{write-lockholders}$) and T' is also the retainer (holder) of conflicting locks, then T' must be an ancestor of T or vice-versa. According to the locking rules, the scheduler grants a lock to T only if all the holders of conflicting locks (T' here) are the ancestors of T . Hence, 1(a) holds, and similarly, 1(b) also holds.

The following property shows which transactions hold locks after a schedule of the DM for the data object X .

Property 2: Let γ be a well-formed schedule of the DM for the data object X . Suppose that γ can leave DM in state s . Let T be an access to the DM such that $\text{REQUEST-COMMIT}(T, v)$ occurs in γ and T is not an orphan at X in γ . Let T' be the highest ancestor of T such that T is visible at X to T' in γ . Then the following holds:

- (a) If T is a write access then T' must be a member of $s.\text{write-lockholders}$.
- (b) If T is a read or transfer access then T' must be a member of $s.\text{read-lockholders}$.
- (c) If T is prewrite-access then T' must be a member of $s.\text{prewrite-lockholders}$.

Proof 2(a): Notice that $\text{REQUEST-COMMIT}(T, v)$ occurs in γ and T' is the highest ancestor of T such that T is visible at X to T' in γ . Therefore, according to the locking rules, T' must have inherited the write-lock from the write access T on its commit. Hence, if T is a write access then $T' \in s.\text{write-lockholders}$, and 2(a) holds. Similarly, 2(b) and 2(c) also hold.

The following property asserts that each prewrite access must be preceded by the corresponding write access.

Property 3: Let γ be a well-formed schedule of a DM. Suppose that γ can leave DM in state s . If $\text{REQUEST-COMMIT}(T, v)$, where T is a write access, has occurred in γ , then it must be preceded in γ by $\text{REQUEST-COMMIT}(T', v)$ where T' is a prewrite access.

Proof: Note the following:

- (a) The prewrite-buffer's value in state s is null if one of the following holds:
 - (a.1) γ has no $\text{REQUEST-COMMIT}(T, v)$ where T is a prewrite access, or an $\text{INFORM-ABORT-AT}(X)$ of U has occurred in γ where U is the parent of T .
 - (a.2) A $\text{REQUEST-COMMIT}(T, v)$ has occurred in γ where T is a write access.
- (b) The prewrite-buffer's value in state s is non-null if one of the following holds:
 - (b.1) A $\text{REQUEST-COMMIT}(T, v)$, where T is a prewrite access, must have occurred in γ and no $\text{INFORM-ABORT-AT}(X)$ of U in γ has occurred where U is the parent of T .

(b.2) No REQUEST-COMMIT(T' , v), where T' is the corresponding write access, has occurred in γ .

The preconditions of REQUEST-COMMIT(T , v), where T is a write access, check that the prewrite-buffer's value must be non-null. That is, a REQUEST-COMMIT(T' , v), where T' is a prewrite access, must have occurred in γ . This also ensures that no other REQUEST-COMMIT(T'' , v), where T'' is some write access, has occurred in γ , i.e., (b.2) holds. Since a REQUEST-COMMIT(T , v) has occurred in γ , where T is a write access, prewrite-buffer's value must be null in the postcondition. This will happen only if (a.2) holds. Hence, on combining (a.2) and (b.2), the property holds.

The following property asserts that the effects of some transactions are durable in case of a transaction abort or system failure.

Property 4: Let γ be a generic object well-formed schedule of a DM. Suppose that γ can leave DM in state s . If a REQUEST-COMMIT(T , v) has occurred in γ where T is a write access then REQUEST-COMMIT(T , v) is durable. Similar assertions hold for prewrite-RM and for the ancestors of prewrite-RM and write access subtransactions.

Proof: Suppose a REQUEST-COMMIT(T , v) where T is a write access has occurred in γ . Now the precondition for REQUEST-COMMIT(T , v) asserts that the write-value must be logged on the stable storage. Also, in the postcondition, the information that a lock is granted to T is also logged. Moreover, the postconditions of the INFORM-COMMIT-AT(X) of T , assert that commit information is also logged. These conditions together ensure the durability of REQUEST-COMMIT(T , v) operation in case of a system crash.

The following property shows that when an access T' occurs, all prior conflicting accesses must either be local orphans or visible to T' .

Property 5: Let γ be a generic object well-formed schedule of a DM. Suppose distinct events $\pi = \text{REQUEST-COMMIT}(T, v)$ and $\pi' = \text{REQUEST-COMMIT}(T', v')$ occur in γ , where T and T' conflict. If π precedes π' in γ then either T is a local orphan in γ' , or T is visible to T' in γ' where γ' is the prefix of γ upto π' .

Proof: If T and T' are two conflicting access transactions and T has committed, then T' can commit only if one of the following holds:

1. The l.c.a.(T , T') has acquired the lock, and therefore, T' can access the object according to the definition of T committed to T' .
2. Some ancestor of T is aborted such that the effects of T can be discarded, and therefore, T will no longer affect T' .

Now since T and T' have committed in γ , and T has committed before T' , then the only possibilities are either 1 or 2, and hence, the property holds.

The following property characterizes the write-buffer component of the state and shows that write-buffer(T) reflects the effects of all transactions that are visible to T . This property from [FLMWb] holds in our model and can be proved along lines similar, consequently the proof is omitted.

Property 6: Let γ be a well-formed schedule of a DM and let s be a state of DM for the data object X . If T is a transaction that is not an orphan at X in γ , then $d = \text{essence}(\text{visible}_X(\gamma, T))$ is a well-formed schedule of the basic object X . Furthermore, when d is applied to an initial state of X , it can leave X in the state $s.\text{write-buffer}(T')$ where T' is the least ancestor of T such that $T' \in s.\text{write-lockholders}$.

The following property is a consequence of the previous property and explains how a data manager automaton is a resilient variance of the basic object X .

Property 7 [18]: Let γ be a well-formed schedule of the DM for the data object X , and T is not an orphan at X in γ . Then $\text{visible}_X(a, T)$ is a well-formed schedule of the basic object X .

Property 8 [18]: Let γ be a concurrent schedule, T a transaction that is not an orphan in γ and X is an object, then $\text{visible}(\alpha, T) \mid X$ is a well-formed schedule of the object X .

The properties given below establish the invariants regarding the buffer management and checkpoint activities during normal operations.

Property 9: Let γ be a well-formed schedule of the DM for the data object X and let s be the state of the DM after γ . Let $\text{REQUEST-COMMIT}(T, v)$ be the last committed write access that updated the value of the object X before the last BGN-CHK-POINT operation. Then $s.\text{write-buffer}(T') = \text{write-value}(T)$, where T' is the least ancestor of T such that $T' \in s.\text{write-lockholders}$.

Proof: In Property 6, we proved that the write-buffer has the value which reflects the effect of all transactions that are visible to T such that if T' is the least ancestor of T then $T' \in s.\text{write-lockholders}$. Because of the above and due to the fact that a BGN-CHK-POINT operation transfers the value of the write-buffer to the stable-db and sets the write-buffer's value to null, the property holds.

Property 10: Let γ be a well-formed schedule of the DM for the data object X and let s_i be a state of the DM such that (s_{i-1}, π_i, s_i) is a step of the DM. Let π_i be a $\text{REQUEST-COMMIT}(T, v)$ where T is a transfer access and $\pi_{i-1} \neq \text{BGN-CHK-POINT}$. Then $\text{stable-db}(X) = s_{i-1}.\text{write-buffer}(T')$, and $\text{stable-db}(X).\text{LSN} = s.\text{write-buffer}.\text{LSN}$ where T' is the least ancestor of T such that $T' \in s_{i-1}.\text{write-lockholders}$.

Proof: A write-buffer is updated by a $\text{REQUEST-COMMIT}(T'', v)$ where T'' is a write access operation. Notice that a transfer access T is invoked under a daemon transaction associated with the write-TM. Let T' be the l.c.a. of T and T'' such that T'' has committed before π_i . By the postcondition of INFORM-COMMIT of T'' , T' has inherited the write-lock and therefore, T can get the read-lock. Hence, $\text{stable-db}(X) = s_{i-1}.\text{write-buffer}(T')$ and also $\text{stable-db}(X).\text{LSN} = s.\text{write-buffer}.\text{LSN}$.

Property 11: Let γ be a well-formed schedule of the DM for the data object X and let (s_{i-1}, π_i, s_i) be a step of the DM. If $\pi_i = \text{BGN-CHK-POINT}$ and there is no j such that $\pi_j >$

π_i and $\pi_j = \text{REQUEST-COMMIT}(T, v)$, where T is a write-access in γ , then $s.\text{write-buffer}(T') = \text{null}$ where T' is the least ancestor of $T \in s.\text{write-lockholders}$ and $\text{stable-db}(X) = s_{i-1}.\text{write-buffer}(T')$.

Proof: By Property 9, the write-buffer keeps the value of the data object until a checkpoint operation occurs, and since there does not exist a $\pi_j = \text{REQUEST-COMMIT}(T, v)$, where T is a write-access in γ , $s.\text{write-buffer}(T') = \text{null}$ and $\text{stable-db}(X) = s_{i-1}.\text{write-buffer}(T')$. Note that all such DMs will require no redo in case of a system crash. Hence, the property holds.

Property 12: Let γ be a well-formed schedule of the DM for the data object X . Let s_i be a state of the DM, then $s_i.\text{write-buffer.LSN} \geq \text{stable-db}(X).\text{LSN}$.

Proof: The only operation which changes the LSN of the $\text{stable-db}(X)$ is a transfer access. If a transfer access has committed in γ in state s_i then $s_i.\text{write-buffer.LSN} = \text{stable-db}(X).\text{LSN}$. Otherwise, if there is no transfer access after the last committed write access, then $s_i.\text{write-buffer.LSN} > \text{stable-db}(X).\text{LSN}$. Hence, the property holds.

Property 13: Let γ be a well-formed schedule of the DM for the data object X . Let (s_{i-1}, π_i, s_i) be a step of the DM where $\pi_i = \text{CREATE}(T)$ and T is not an orphan, then $T \in \text{TT}$.

Proof: One of the postconditions of a CREATE operation for a DM asserts that an entry must be in the transaction table signifying that T is active. Hence, the property follows.

Property 14: Let γ be a well-formed schedule of the DM for the data object X . Let (s_{i-1}, π_i, s_i) be a step of the with $\pi_i = \text{REQUEST-COMMIT}(T, v)$ and T being a write access, prewrite-RM or their ancestors, then $T \in \text{LT}$, TT and T also $\in \text{DOT}$ if T is a write or prewrite access.

Proof: The pre and post conditions of the given access transactions for an object automaton assert that as soon as a $\text{CREATE}(T)$ occurs, an entry $\langle T, \text{active} \rangle$ is made into the transaction table (see Property 13). Also, a corresponding entry is made into the lock table if the transaction holds any lock. If it is an update transaction, a corresponding entry is also made into the dirty object table. Also, an entry from the TT and LT is removed only on receiving the corresponding INFORM-COMMIT operation. A prewrite entry from the DOT is removed on receiving the REQUEST-COMMIT of the corresponding write operation or an abort of the prewrite access, whereas a write entry is removed on the REQUEST-COMMIT of transfer access or on END-CHK-POINT operation. Therefore, the property holds.

7. CORRECTNESS

In the previous section, we have proved some properties of the states of the data objects with respect to different transactions which established that pre and post conditions capture the algorithm's behavior at the object level. We now give the serial correctness proof of the concurrency control scheme. As defined in [2], a sequence of operations α of a target system is serially correct for a transaction T provided the

projection of α onto T is identical to the projection of T onto some serial schedule β of the corresponding serial system.

The correctness proof of the algorithm is as follows. First, an appropriate serial system is constructed. Next, we take a concurrent schedule α of the target system and extract a subsequence of operations from α whose effects might have been detected by T , called $\text{visible}(\alpha, T)$, and show its equivalence to a schedule obtained by the Moss's two phase locking and the serial system.

We do not deal with all the operations at each object, but only those that are in some sense visible to (hence affect) a particular transaction T . A transaction T' can affect another transaction T in the following way. If T' is an ancestor of T then T' can affect T by passing information down the transaction tree via invocations. A transaction T' that is not an ancestor of T can affect T through COMMIT action for T' and/or through the COMMIT of all the ancestors of T' up to the level of the least common ancestor with T depending on the type of transactions since information can be propagated from T' to the least common ancestor via lock inheritance actions at the time of commit, and from there down to T via invocations. A transaction T' that is not an ancestor of T can affect T by accessing an object that is later accessed by T . In most of the usual transaction processing algorithms, this is allowed to occur if there are intervening COMMIT actions for all ancestors of T' up to level of the least common ancestor with T .

The $\text{visible}(\alpha, T)$ formally captures the sequence of operations visible in the above sense. Once such a subsequence of operations is obtained, we show that the order of operations in the subsequence can be rearranged to give a schedule obtained by Moss's two phase locking and a serial schedule both having the same sequence of REQUEST-COMMIT of conflicting access subtransactions. That is, the rearrangement of operations should always be such that it transforms one sequence into another thereby ensuring that the commit ordering of conflicting access subtransactions are maintained in both sequences. In other words, the rearrangement of operations of objects are such that the difference between the orders is not detectable by any later operations of that object. We then prove that there always exists a serial schedule β that is write-equivalent to $\text{visible}(\alpha, T)$. That is, we explicitly construct a serial schedule β such that the projection of a transaction T' on the serial schedule β is same as its projection on $\text{visible}(\alpha, T)$. If β is such a serial schedule and T is visible to T' then $\text{visible}(\alpha, T) \upharpoonright T' = \alpha \upharpoonright T'$ and thus, $\alpha \upharpoonright T' = \beta \upharpoonright T'$, i.e., the serial schedule β is consistent with the local schedules of every transaction T' .

7.1 Construction of a Serial System

The composition of transactions, objects and the serial scheduler for a given system type is called a serial system and its operations and schedules are called serial operations and serial schedules, respectively. The serial system corresponding to our nested transaction concurrency control system will have the same set of transactions as defined in our concurrent model. The serial scheduler and the objects in the serial system are as described in [18] except that the objects receive two more types of access operations called prewrite and transfer accesses. A prewrite access operation will have the same pre and post conditions as those given in the case of a write access, whereas a transfer access will have pre and postconditions like that of a read access. The objects also receive two

more operations for checkpoint activity, namely BGN-CHK-POINT as an input operation for initiating checkpoint, and END-CHK-POINT as an output operation which announces the end of the checkpoint activity. The DM in serial order receives these operations when no other transaction is active.

7.2 Structure of a Schedule and Rearrangement of Operations

Any schedule in our scheme will consist of operations of user-visible transactions (non-access), access transactions and the scheduler. In our model, we have the following pairs of conflicting operations, namely read-write, read-prewrite, write-prewrite, write-write and prewrite-prewrite. The locking protocols are such that the conflicting locks are never held except when one transaction is the ancestor of the other. Also, as in any nested transaction model, a parent transaction cannot commit until its children are terminated (either committed or aborted).

We have given an algorithm that performs concurrency control. In order for the algorithm to work correctly, we require that operations whose locks do not conflict have the same effect regardless of the order in which they occur.

According to the locking protocols, some transactions are allowed to release their locks on commit to their least common ancestor prior to the commit of transactions up to the l.c.a. Specifically, prewrite-RMs, write accesses and their ancestors, on commit, can release their locks to the l.c.a. of the other waiting accesses prior to the commit of all the transactions up to the l.c.a. This will allow non-access subtransactions in the hierarchy of siblings of the l.c.a. to commit in an interleaving fashion. Also, the access subtransactions can commit in an interleaved fashion with the ancestors of their siblings.

In any serial schedule, the transactions commit ordering is induced by depth first traversal from leaf to root order (i.e., in ascending order) without interleaving. To begin the process of obtaining such a serial order, we rearrange some of the interleaved operations in the schedule obtained by our locking protocols. These rearrangements of operations are such that the well-formed conditions of the operations in the new sequence are maintained. The rearrangements of operations permitted by our locking protocols are as follows:

- (a) Rearrange the order of two events of different transactions or objects, and also rearrange the order of events of a single object provided both are not REQUEST-COMMIT for conflicting accesses. Thus, the order of conflicting operations will remain the same in the two schedules. This rearrangement rule is same as given in [18] with respect to Moss's two phase locking.
- (b) Rearrange the order of events including the REQUEST-COMMIT operations of non-access subtransactions sharing the same l.c.a. This process is to be started from one level above the leaf level and follows up to the root.
- (c) Rearrange the order of events including the REQUEST-COMMITs of access subtransactions with the ancestors of their siblings such that the commit order of conflicting access subtransactions are not changed. This process is also to be started from one level above the leaf level and follows up to the root.

Example: Consider the following subsequence of operations visible(α , T) of schedule a where T_3 and T_5 are prewrite and write accesses, respectively, and T_4 is a read access as shown in Fig. 2. Let X be the data object accessed by these subtransactions.

```

CREATE(T) REQUEST-CREATE( $T_1$ ) REQUEST-CREATE( $T_2$ ) CREATE( $T_2$ )
CREATE( $T_1$ ) REQUEST-CREATE( $T_3$ ) REQUEST-CREATE( $T_4$ ) CREATE( $T_4$ )
REQUEST-CREATE( $T_4$ ) CREATE( $T_3$ ) REQUEST-COMMIT( $T_3$ ) COMMIT( $T_3$ )
INFORM-COMMIT-AT-(X) OF  $T_3$  REQUEST-CREATE( $T_5$ ) CREATE( $T_5$ )
REQUEST-COMMIT( $T_4$ ) COMMIT( $T_4$ ) INFORM-COMMIT-AT-(X) OF  $T_4$ 
REQUEST-COMMIT( $T_5$ ) COMMIT( $T_5$ ) INFORM-COMMIT-AT-(X) OF  $T_5$ 
REQUEST-COMMIT( $T_2$ ) COMMIT( $T_2$ ) INFORM-COMMIT-AT-(X) OF  $T_2$ 
REQUEST-COMMIT( $T_1$ ) COMMIT( $T_1$ ) INFORM-COMMIT-AT-(X) OF  $T_1$ 
REQUEST-COMMIT(T) COMMIT(T) INFORM-COMMIT-AT-(X) OF T.

```

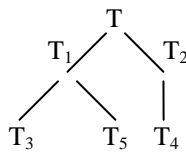


Fig. 2. Example tree.

Observe that the ordering of transactions in the above subsequence has a different order compared to any schedule of a serial system. However, an equivalent serial schedule can be obtained by applying the rearrangement rules given above as follows:

1. Rearrange the REQUEST-COMMITs of T_4 and T_5 using rule (a) The REQUEST-COMMIT of a read subtransaction after a prewrite operation, but before the corresponding write, can be interchanged with REQUEST-COMMIT of the write operation. This is because the values returned by a data object in response to a read after the commit of a prewrite, but before the corresponding write operation, will be the same as those returned by any read after the commit of the corresponding write operation. Note that once T_3 is committed, it releases the lock to T according to the locking protocols, and hence, T_4 can get the read-lock before T_5 acquires the write-lock. Since the lock is released to T, no transaction outside the hierarchy of T can get the lock. That is, there will not be any other access between T_4 and T_5 . This new sequence of operations is permitted because once T_3 is committed, T_5 is also eligible to get the lock, and hence its commit can come before T_4 . Therefore, commit ordering of such leaf level transactions can be rearranged. However, this disturbs well-formedness since COMMIT of T_4 and T_5 cannot occur before their REQUEST-COMMITs. Since our locking protocols produce only well-formed schedules, we also rearrange corresponding operations. As the REQUEST-COMMITs of other operations are not disturbed, the new arrangement is allowed by the locking protocols. This new rearrangement of operation is well-formed since the REQUEST-COMMITs of parent transactions T_1 and T_2 occur after the commits of their subtransactions (T_3 , T_5 , and T_4).

Note that altering the commit order of read and write subtransactions make the prewrite operation redundant, which shows that the prewrite operation is used only to increase availability.

The resulting schedule is as follows:

```
CREATE(T) REQUEST-CREATE(T1) REQUEST-CREATE(T2) CREATE(T2)
CREATE(T1) REQUEST-CREATE(T3) REQUEST-CREATE(T4) CREATE(T4)
REQUEST-CREATE(T4) CREATE(T3) REQUEST-COMMIT(T3) COMMIT(T3)
INFORM-COMMIT-AT-(X) OF T3 REQUEST-CREATE(T5) CREATE(T5)
REQUEST-COMMIT(T5) COMMIT(T5) INFORM-COMMIT-AT-(X) OF T5
REQUEST-COMMIT(T4) COMMIT(T4) INFORM-COMMIT-AT-(X) OF T4
REQUEST-COMMIT(T2) COMMIT(T2) INFORM-COMMIT-AT-(X) OF T2
REQUEST-COMMIT(T1) COMMIT(T1) INFORM-COMMIT-AT-(X) OF T1
REQUEST-COMMIT(T) COMMIT(T) INFORM-COMMIT-AT-(X) OF T2.
```

2. Rearrange the REQUEST-COMMITs of T₂ and T₁ using rule (b). The REQUEST-COMMITs of T₂ and T₁ do not conflict because they are non-access subtransactions, and T₄ receives the value updated by T₃ irrespective of the order in which T₁ and T₂ commit. Hence, their order can be changed. As T₁ and T₂ are non-access subtransactions, they are permitted to commit in either order by the locking protocols. To maintain well-formedness, interchange of COMMIT and INFORM-COMMIT operations of T₂ and T₁. Since the ordering of other REQUEST-COMMIT operations are not changed, this is allowed by the locking protocols. The resulting schedule is:

```
CREATE(T) REQUEST-CREATE(T1) REQUEST-CREATE(T2) CREATE(T2)
CREATE(T1) REQUEST-CREATE(T3) REQUEST-CREATE(T4) CREATE(T4)
REQUEST-CREATE(T4) CREATE(T3) REQUESTVCOMMIT(T3) COMMIT(T3)
INFORM-COMMIT-AT-(X) OF T3 REQUEST-CREATE(T5) CREATE(T5)
REQUEST-COMMIT(T5) COMMIT(T5) INFORM-COMMIT-AT-(X) OF T5
REQUEST-COMMIT(T4) COMMIT(T4) INFORM-COMMIT-AT-(X) OF T4
REQUEST-COMMIT(T1) COMMIT(T1) INFORM-COMMIT-AT-(X) OF T1
REQUEST-COMMIT(T2) COMMIT(T2) INFORM-COMMIT-AT-(X) OF T2
REQUEST-COMMIT(T) COMMIT(T) INFORM-COMMIT-AT-(X) OF T3.
```

3. Rearrange the REQUEST-COMMITs of T₄ and T₁ by applying rule (c). The REQUEST-COMMITs of T₄ and T₁ do not conflict because the value received by T₄ is the value updated by T₃ and the commit ordering of T₃ and T₄ is not changed. But we cannot move the REQUEST-COMMIT of T₄ as it is an access transaction and therefore, may be in conflict if there is any other access committed after T₄. Therefore, we move the REQUEST-COMMIT of T₁ to just after the INFORM-COMMIT of T₅. This is because T₅ is the last committed child transaction of T₁ and therefore, REQUEST-COMMIT of T₁ cannot occur before T₅. This new rearrangement of operations is permitted by our locking protocols as T₁ can commit before T₄, and T₄'s commit order is not disturbed. The rearrangement of operations should always be such that the commit of subtransactions should occur before the commit of their parent transaction. However, this rearrangement disturbs the

well-formedness since COMMIT and INFORM-COMMIT of T_1 occur after T_4 . Therefore, we place COMMIT and INFORM-COMMIT of T_1 just after its REQUEST-COMMIT as per the locking rules. The new arrangement of operations will not disturb the well-formedness conditions of other operations since their respective ordering is not altered.

The new arrangement of operations results in the following schedule:

```
CREATE(T) REQUEST-CREATE( $T_1$ ) REQUEST-CREATE( $T_2$ ) CREATE( $T_2$ )
CREATE( $T_1$ ) REQUEST-CREATE( $T_3$ ) REQUEST-CREATE( $T_4$ ) CREATE( $T_4$ )
REQUEST-CREATE( $T_4$ ) CREATE( $T_3$ ) REQUEST-COMMIT( $T_3$ ) COMMIT( $T_3$ )
INFORM-COMMIT-AT-(X) OF  $T_3$  REQUEST-CREATE( $T_5$ ) CREATE( $T_5$ )
REQUEST-COMMIT( $T_5$ ) COMMIT( $T_5$ ) INFORM-COMMIT-AT-(X) OF  $T_5$ 
REQUEST-COMMIT( $T_1$ ) COMMIT( $T_1$ ) INFORM-COMMIT-AT-(X) OF  $T_1$ 
REQUEST-COMMIT( $T_4$ ) COMMIT( $T_4$ ) INFORM-COMMIT-AT-(X) OF  $T_4$ 
REQUEST-COMMIT( $T_2$ ) COMMIT( $T_2$ ) INFORM-COMMIT-AT-(X) OF  $T_2$ 
REQUEST-COMMIT(T) COMMIT(T) INFORM-COMMIT-AT-(X) OF T
```

Assertion 1: The rearrangement of operations as per rule (a), (b) and (c) satisfies the well-formedness conditions and will not introduce any new conflicts.

Proof: The well-formedness conditions of the operations given in [18] are preserved in (a), (b) and (c). This can be argued as follows:

The projections of operations of every transaction and object onto the new sequence, whose operations are not rearranged, are well-formed. That is, $\alpha|T$ and $\alpha|X$ are well-formed if the operations of X and T are not rearranged. If the well-formedness conditions of operations of transactions and objects involved are disturbed, we again rearrange such operations to make the new arrangement well-formed.

The commit order of conflicting access subtransactions are not altered in (a), (b) and (c). Thus, no new conflict is introduced.

In our model subtransactions at leaf level define the order of accesses on the DM. That is, a computation of a set of transactions consists of their access operations as ordered by their execution (REQUEST-COMMIT) on the DM and not as ordered by their invocation or invocation and completion of the higher-level transactions. Therefore, the rearrangement of operations in (b) and (c) will not give rise to conflicting situations.

Because commit orders of conflicting access subtransactions are not altered, and due to the fact that their ancestors are non-access subtransactions and correspond to only procedural nesting in our model, the REQUEST-COMMIT operations of the ancestors of two access subtransactions can always be rearranged. Hence, the rearrangement of operations in (b) and (c) will not introduce any new conflicts.

We now show that:

1. The rearrangement of operations (a), (b) and (c) gives rise to a schedule obtained by Moss's two phase locking which in turn shows that the relaxation of some of the rules of Moss's two phase locking in our model does not introduce any new conflicts but can help in achieving more availability.

2. The rearrangement of operations transfer one sequence into another write-equivalent serial schedule.

7.4 Relationship With Moss's Two Phase Locking and Proof of Correctness

Recall that our locking protocols are extensions of Moss's two phase locking rules [1]. Our model permits some subtransactions to release their locks to their l.c.a. before the commit of transactions up to the l.c.a. This allows some access subtransactions sharing the same l.c.a. to commit in an interleaved fashion. In section 7.2, we gave rules for rearrangement of some of the operations such that new ordering introduces no new conflict. Now, we show that the rearrangement of operations in the schedule of our model has the same ordering of operations as in any schedule obtained by Moss's two phase locking. This establishes that the modifications incorporated in our locking scheme over Moss's two phase locking increase availability and they do not introduce any new conflicts.

Assertion 2: The rearrangement rules for operations given in section 7.2. always result in a schedule which is write-equivalent to the one obtained by Moss's two phase locking.

Proof: First, we observe the structure of a schedule under Moss's two phase locking.

A concurrent schedule under Moss's two phase locking consists of operations of user-visible transactions, access subtransactions and the scheduler operations. Since concurrent operations are controlled by Moss's two phase locking, the lock release operations occur in order from leaf to root (i.e., in ascending order) such that the commit ordering of operations depends on the release of locks from a the child to the parent. In this model, a subtransaction releases its lock to its parent, and so on, to restrict the visibility of updated values within its parent's hierarchy in order to avoid inconsistencies in case of transaction aborts at higher-level. This will avoid the intervening commit of subtransactions under the hierarchy of siblings. Formally, a well-formed schedule under Moss's two phase locking has the following properties [18]:

- a1. It assumes that conflicting locks are never held by transactions except when one transaction is an ancestor of the other. This condition is enforced when locks are granted, and preserved thereafter.
- a2. A parent transaction can be terminated (committed or aborted) only after the termination of its children. That is, if there is a commit of T, then there cannot be an abort or commit of T', where T' is a child of T. This property can be stated formally as follows : Let $\alpha = \alpha'\pi$ be a concurrent schedule where π is REQUEST-CREATE(T'), COMMIT(T'), or ABORT(T') for a child T' of T. Then α' does not contain a COMMIT(T) event.
- a3. A transaction T is visible at X to transaction T' if an INFORM-COMMIT-AT-(X) OF U event occurs for every $U \in \text{ancestors}(T)\text{-ancestors}(T')$, arranged in ascending order (i.e., INFORM-COMMIT for parent(U) is preceded by that for U). The INFORM-COMMIT operations of transactions will always be such that every INFORM-COMMIT an INFORM-COMMIT of a descendant of all transactions preceding it in the sequence and an ancestor of all transactions

following it in the sequence. The last INFORM-COMMIT will always be of the root. Note that any INFORM-COMMIT is preceded by the COMMIT(T) event (by the generic scheduler preconditions given in [18]).

- a4. If γ is a schedule obtained by Moss's two phase locking and T is not an orphan then $\text{visible}(\gamma, T) \mid X$ is a well-formed schedule of operations of X.

Now we observe that the above properties are also preserved from the schedule obtained by the rearrangement of operations allowed by our locking protocols.

Properties a1 and a2 follow from the locking rules. Property a3 is satisfied, due to the rearrangement of operations (a), (b) and (c) given in section 6.2. The property a4 is satisfied since Property a3 is satisfied, and therefore, REQUEST-COMMIT and INFORM-COMMIT operations of non-access transactions operating on X are rearranged in ascending order.

Since all the properties of Moss's two phase locking are preserved by schedules obtained using our locking protocols, they are also write-equivalent.

Now, the following lemma from [18] establishes the fact that there is a serial schedule β that is write-equivalent to $\text{visible}(\alpha, T)$. Since we have proved that the rearrangement of operations results in the ordering of operations as in some schedule of Moss's two phase locking, the serial schedule can be constructed on lines similar to that given in [18]. Therefore, all the cases follow immediately from [18]. Thus, we state the lemma without proof.

Lemma 2: Let α be a concurrent schedule, and T any transaction that is not an orphan in α . Then there is a serial schedule β that is write-equivalent to $\text{visible}(\alpha, T)$.

We now state that system B is serially correct for every transaction that is not an orphan with the help of the following theorem [18]:

Theorem: Every concurrent schedule is serially correct for every non-orphan transaction.

Proof: Let α be a concurrent schedule, and T any transaction that is not an orphan in α . By the previous lemma, we see that there is a serial schedule β that is write-equivalent to $\text{visible}(\alpha, T)$. Also, we know the fact that $\text{visible}(\alpha, T) \mid T'$ is equal to $\alpha \mid T'$ if T' is visible to T in α . Therefore, $\alpha \mid T = \text{visible}(\alpha, T) \mid T$ and by definition of write-equivalence, we have $\text{visible}(\alpha, T) \mid T = \beta \mid T$.

8. CONCLUSIONS

In this paper we discussed an open and safe nested transaction model and the concurrency control algorithm. We logically implemented our model using an I/O automaton model with the aim of proving formal correctness. A sketch of the recovery algorithm is also given. Information required to deal with system restart is taken into account during the specification of pre and post conditions. We have proved that our concurrency control algorithm is serially correct. We mapped our system to that of Moss's two phase locking system to show the relationships in the correctness proof of the two algorithms.

With regard to future work, our transaction model needs to be extended in the context of orthogonally persistent programming languages, where serious problems arise unless all computation exists within a transactional context, although this restricts concurrency. We are exploring this issue further. Our transaction model has also been used in mobile computing, and its performance evaluation has also been reported in [38], but not in a nested transaction environment.

REFERENCES

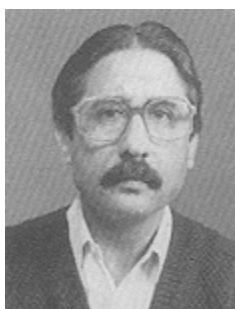
1. J. E. B. Moss, "Nested transactions: An approach to reliable distributed computing," Ph.D. Thesis, and Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA., April, 1981.
2. N. Lynch and M. Merrit, "Introduction to the theory of nested transactions," *Theoretical Computer Science*, Vol. 62, 1988, pp. 123-185.
3. B. Liskov, "Distributed computing in Argus," *Communication of ACM*, Vol. 31, 1988, pp. 300-312.
4. W. Schaad, H.-J. Schek, and G. Weikum, "Implementation and performance of multi-level transaction management in a multidatabase environment," RIDE-DOM 1995, pp. 108-115.
5. W. Kim, R. Lorie, D. McNabb, and W. Plouffe, "A transaction mechanism for engineering design databases," in *Proceedings of the 10th International Conference on Very Large Databases*, VLDB Endowment, 1984, pp. 355-362.
6. H. F. Korth, W. Kim, and Bancilhon, "On long-duration CAD transactions," *Information Science*, Vol. 46, 1990, pp. 73-107.
7. W. E. Weihl, "Specifications and implementation of atomic data types," Ph.D. Thesis, and Technical Report MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, MA., March, 1984.
8. C. Beeri, P. A. Bernstein, and N. Goodman, "A model for concurrency in nested transaction system," *Journal of the ACM*, Vol. 36, 1989, pp. 230-269.
9. G. Weikum, "Principles and realization strategies of multi-level transaction management," *ACM Transaction on Database System*, Vol. 16, 1991, pp. 132-180.
10. J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzulo, and I. Traiger, "The recovery manager of the system R database manager," *ACM Computing Surveys*, Vol. 13, 1981, pp. 223-244.
11. J. E. B. Moss, N. Griffith, and M. Graham, "Abstraction in concurrency control and recovery management (revised)," Technical Report COINS 86.20, Dept. of Computer Science, University of Massachusetts at Amherst, May, 1986.
12. P. Muth, T. C. Rakow, G. Weikum, P. Brossler, and C. Hasse, "Semantic concurrency control in object-oriented database systems," in *Proceedings of the 9th International Conference on Data Engineering*, 1993, pp. 233-242.
13. R. F. Resende, D. Agrawal, and A. E. Abbadi, "Semantic locking in object oriented database systems," Technical Report TRCS 94-01, Dept. of Computer Science, University of California at Santa Barbara, 1994.
14. S. K. Madria, "Concurrency control and recovery algorithms in nested transaction environment and their proofs of correctness," Ph.D. Thesis, Indian Institute of Technology, Delhi, India, 1995.

15. S. K. Madria and B. Bhargava, "System defined prewrites to increase concurrency in databases," *First East-European Symposium on Advances in Databases and Information Systems*, 1997, pp. 18-22.
16. H. F. Korth and G. Speegle, "Long duration transactions in software design projects," in *Proceedings of 6th International Conference on Data Engineering*, IEEE, 1990, pp. 568-574.
17. M. A. Tubaishat, S. K. Madria, and B. Bhargava, "Performance evaluation of linear hash structures in a nested transaction environment," *Journal of Systems and Software*, to appear on 2002.
18. A. Fekete, N. Lynch, M. Merrit, and W. Weihl, "Nested transactions and read/write locking," in *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, 1987, pp. 97-111.
19. A. Fekete, N. Lynch, M. Merrit, and W. Weihl, *Atomic Transactions*, Morgan-Kaufmann, 1993.
20. N. Lynch, "Concurrency control for resilient nested transactions," *Advances in Computing Research*, Vol. 3, 1986, pp. 335-376.
21. D. P. Reed, "Naming and synchronization in a decentralized computer system," Ph.D. Thesis, and Technical Report MIT/LCS/TR-205, MIT Laboratory for Computer Science, MA., 1978.
22. J. Aspnes, A. Fekete, N. Lynch, M. Merrit, and W. Weihl, "A theory of timestamp based concurrency control for nested transactions," in *Proceedings of 14th International Conference on Very Large Databases*, 1988, pp. 431-444.
23. A. Fekete, N. Lynch, M. Merrit, and W. Weihl, "Commutativity-based locking for nested transactions," *Journal of System Sciences*, Vol. 41, 1990, pp. 65-156.
24. W. E. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Transaction on Computers*, Vol. 37, 1988, pp. 1488-1505.
25. A. Fekete, N. Lynch, and W. Weihl, "A serialization graph construction for nested transactions," in *Proceedings of ACM Symposium on Principles of Database Systems*, 1990.
26. J. Gray, W. Lorie, G. Putzolu, and I. Traiger, "Granularity of locks and degree of consistency in a shared database," in *Modeling in Database Management Systems*, G. Nijssen ed., 1976, pp. 365-394.
27. J. K. Lee and A. Fekete, "Multi-granularity locking for nested transaction systems," in *Proceedings of Mathematical Fundamentals of Database Systems '91*, LNCS 495, Springer Verlag, 1991, pp. 160-172.
28. J. K. Lee and A. Fekete, "Predicate locking for nested transaction systems," in *Proceedings of Australian Database Research Conference*, 1992, pp. 217-231.
29. J. K. Lee, "Precision locking for nested transaction systems," in *Second International Conference on Information and Knowledge Management (CIKM'93)*, 1993, pp. 674-683.
30. D. Gifford, "Weighted voting for replicated data," in *Proceedings of the 7th Symposium on Operating Systems Principles*, 1979, pp. 150-162.
31. K. Goldman and N. Lynch, "Nested transactions and quorum consensus," *ACM TODS*, 1994, pp. 537-585.
32. S. K. Madria, S. N. Maheshwari, and B. Chandra, "Virtual partition algorithm in a nested transaction environment and its correctness," *Information Sciences*, Vol. 137, 2001, pp. 211-244.

33. A. Fu and T. Kameda, "Concurrency control of nested transactions accessing B-trees," in *Proceedings of 8th ACM Symposium on Principles of Database Systems*, 1989, pp. 270-285.
34. S. K. Madria, S. N. Maheshwari, and B. Chandra, "Formalization and correctness of a concurrent linear hash structure algorithm using nested transactions and I/O automata," *DKE*, Vol. 37, 2001, pp. 139-176.
35. S. K. Madria, M. A. Tubaishat, and B. Bhargava, "Multi-level transaction model for semantic concurrency control in linear hash structure," *Information and Software Technology Journal*, Vol. 42, 2000, pp. 445-464.
36. S. K. Madria, S. N. Maheshwari, and B. Chandra, "Formalization and correctness of a concurrency control algorithm for an open and safe nested transaction model using I/O automaton model," in *Proceedings of 8th International Conference on Management of Data (COMAD'97)*, 1997, pp. 140-161.
37. S. K. Madria, S. N. Maheshwari, and B. Chandra, "Formalization and correctness of the recovery algorithm for an open and safe transaction model," *International Journal of Co-operative Information Systems*, Vol. 10, 2001, pp. 1-50.
38. S. K. Madria and B. Bhargava, "A transaction model to improve data availability in mobile computing," *Distributed and Parallel Databases* Vol. 10, 2001, pp. 127-160.



Sanjay Kumar Madria received his Ph.D. in Computer Science from Indian Institute of Technology, Delhi, India in 1995. He is an assistant professor of the department of Computer Science at the University of Missouri-Rolla, USA. Earlier he was Visiting Assistant Professor in the Department of Computer Science, Purdue University, West Lafayette, U.S.A.. He has published more than 70 journal and conference papers in the areas of web warehousing, mobile databases, data warehousing, nested transaction management and performance issues. He has chaired international conferences and workshops, organized tutorials, and has actively served in the program committees of numerous international conferences and has been reviewer for many journals. He participated as panelist in National Science Foundation and Swedish Research Council. He is an IEEE senior member and ACM member.



S. N. Maheshwari received his Ph.D. in Computer Science from Northwestern University, USA. He is currently a Professor of Computer Science and Engineering at Indian Institute of Technology, Delhi, India, where he has held the appointments of Head, Department of Computer Science and Engineering, and Dean, Undergraduate Studies. His research interests range from graph algorithms, computational geometry, distributed and parallel computing, to theory of transaction processing.



B. Chandra received her Ph.D. in Operational Research from Delhi University, India. She is a Professor of computer science in the Department of Mathematics, Indian Institute of Technology, Delhi, India. Her research interests include transaction processing, expert systems and artificial intelligence.