LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-476

HYBRID ATOMICITY FOR NESTED TRANSACTIONS

Alan Fekete
Nancy Lynch
William E. Weihl

October 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Hybrid Atomicity for Nested Transactions

Alan Fekete*[1] and Nancy Lynch**[2] and William E. Weihl***[2]

[1] Department of Computer Science,
University of Sydney, 2006, Australia
[2] MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.

Abstract. This paper defines the notion of *hybrid atomicity* for nested transaction systems, and presents and verifies an algorithm providing this property. Hybrid atomicity is a modular property; it allows the correctness of a system to be deduced from the fact that each object is implemented to have the property. It allows more concurrency than dynamic atomicity, by assigning timestamps to transactions at commit. The Avalon system provides exactly this facility.

Key words: hybrid atomicity, hybrid system, hybrid object, concurrency control, databases, locking, timestamps

1 Introduction

Two-phase locking [4] is probably the most widely used method of concurrency control in transaction systems today. In recent years much research has focused on extending concurrency control methods to take the semantics of the data into account, thus permitting more concurrency by allowing transactions executing commuting operations to run concurrently (e.g., see [9, 14, 17, 16, 15]). Such "logical locking" can be important to avoid concurrency bottlenecks that arise at frequently updated data items (or "hot spots"). For some applications, however, the requirement that

* Supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

** Supported in part by the National Science Foundation under Grants CCR-86-11442 and CCR-89-15206, in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and in part by the Office of Naval Research under Contracts N00014-85-0168 and N00014-91-J-1046.

*** Supported in part by the National Science Foundation under Grant CCR-8716884, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988. Also supported in part by an equipment grant from Digital Equipment Corporation.

non-commuting operations must conflict can hurt performance by restricting concurrency. Recently, Herlihy and Weihl proposed a new technique, based on assigning timestamps to transactions as they commit and propagating the timestamp information to objects, that allows some of the conflicts imposed by commutativity to be eliminated [7, 8]. In this paper we extend their algorithm to accommodate nested transactions, using the framework developed in [5]. Our results show the generality of the framework used here, and also point out the subtleties involved both in defining algorithms for nested transactions and in proving them correct.

Locking algorithms serialize transactions dynamically in the order in which they commit. However, detailed information about the commit order is not usually available to the concurrency control algorithm, particularly in a distributed system; instead, locking makes conservative assumptions about the commit order based on when locks are acquired and released. Thus, commutativity-based algorithms require an operation executed by a transaction to commute with all operations previously executed by other transactions that are still active; this ensures that regardless of the order in which they commit, their operations will be serializable in that order. As Herlihy and Weihl discuss, however, commutativity-based algorithms allow very little concurrency for some applications. For example, the enqueue and dequeue operations on a FIFO queue do not commute, so commutativity-based locking reduces to exclusive locking, preventing one transaction from accessing the queue until the previous one has committed.

Herlihy and Weihl describe hybrid techniques that combine aspects of timestamp-based and locking algorithms. Their algorithm relies on timestamps generated as transactions commit to capture the commit order. Objects learn the exact commit order by being told the timestamps for committed transactions. As discussed in more detail below, this information can be used to relax the constraints imposed by commutativity-based locking by basing the conflict relations on *serial dependency relations*, rather than on commutativity. For example, the enqueue operations on a FIFO queue do not need to depend on each other, so transactions executing enqueue operations can be allowed to run concurrently. The apparent serialization order of the enqueues can be sorted out based on the timestamps generated when transactions commit, so the order of items in the queue can be determined for subsequent dequeues.

In this paper, we show how Herlihy and Weihl's algorithm can be extended to accommodate nested transactions. Nested transactions have been explored in a number of projects (e.g., [13, 12, 10, 3, 1]) for building reliable distributed systems. In a nested transaction system, a transaction can have subtransactions, each of which appears to run atomically within the transaction. Thus, concurrent subtransactions are serializable — they appear to run in some serial order — and recoverable — they appear to execute either completely or not at all. In addition, if a subtransaction aborts, its parent is informed of the abort and can choose to try some alternative action (e.g., in a replicated system).

We give a precise, formal description of the extended algorithm. We use the framework presented in [5] as a basis for this work. This framework provides a rigorous foundation for nested transaction systems based on a formal operational model. Nested transactions introduce a number of subtleties, concerning the precise handling of concurrent subtransactions and of aborts, that require a careful rigorous

treatment.

Our presentation parallels our earlier work on locking algorithms. We describe a system consisting of transactions plus objects, together with a controller that mediates communication between the transactions and the objects. We use the general definition of correctness from our previous work, and define a local property of objects, called *hybrid atomicity*, that is sufficient to guarantee global correctness.[3] (I.e., if each object in a system is hybrid atomic, the system as a whole is correct.) Hybrid atomicity captures the property of an individual object that says that it serializes transactions in the commit order provided to the object by the timestamps generated at commit. Then we show how to extend Herlihy and Weihl's algorithm to handle nested transactions; the resulting object is hybrid atomic.

Introducing a local property such as hybrid atomicity affords important modularity. Each object can be implemented independently, and as long as each is hybrid atomic, the entire system will be correct. Simple concurrency control techniques (e.g., exclusive locking or read-write locking) can be used where the need for concurrency is small, and more complex techniques (e.g., the algorithm described in this paper) can be used in the (usually few) cases where more concurrency is needed. Hybrid atomicity captures the properties of the interactions among objects that are essential for global correctness, in particular, how they agree on a serialization order for transactions.

The Avalon system [3] (built on top of Camelot) has adopted hybrid atomicity for nested transactions as the basis of its operation. The **tid** or transaction identifier generated by the system has a comparison operation that indicates which of two transactions committed first. This information is just what is needed by our algorithm, and thus our algorithm could be used in the Avalon system.

The remainder of this paper is organized as follows. First, in Section 2 we define the model appropriate for a system assigning timestamps at commit time; we also define hybrid atomicity. In Section 3 we present an algorithm that is hybrid atomic. Finally, we conclude with a discussion and some suggestions for further work. In an Appendix, we briefly summarize the earlier work of ours that provides the framework for this paper. Because of length constraints, this paper omits all the proofs of our results. The proofs will appear in [6].

2 Hybrid Atomicity

This section depends on our earlier work, presented as Sections 3 to 5 of [5], and summarized in the Appendix. The development in this section closely parallels that in Section 6 of [5] and also that in [2]. In our presentation we concentrate on those aspects that are different from the previous papers.

We define the system decomposition appropriate for describing hybrid algorithms. Such algorithms are formulated as instances of *hybrid systems*, which are composed of transaction automata, *hybrid object automata* and a *hybrid controller*.

Throughout, we use a totally ordered set $\mathcal{P}$ of *timestamps*. In our development, we will not actually need the set $\mathcal{P}$ to be totally ordered – it will be enough that the

[3] Weihl defined several local properties for single-level transaction systems [17, 16]; the local property defined here generalizes one of those to nested transaction systems.

timestamps assigned to sibling transactions be ordered with respect to each other. However, for simplicity we assume the total ordering. A natural choice for $\mathcal{P}$ is the set of positive integers, or more realistically, the integers less than some (extremely large) maximum.

Hybrid Object Automata A hybrid object automaton H_X for an object name X is an automaton with the following actions, which define its interface to its environment. The input actions are **CREATE**(T), for T an access to X, **INFORM_COMMIT_AT**(X)**OF**(T, p), for $T \neq T_0, p \in \mathcal{P}$, and **INFORM_ABORT_AT**(X)**OF**(T), for $T \neq T_0$. The output actions are **REQUEST_COMMIT**(T,v), for T an access to X and v a value for T. In addition, H_X may have an arbitrary set of internal actions.

The interface of a hybrid object automaton H_X is similar to that of a generic object G_X, as defined in [5] to model data managers that receive requests for data access and information about the completion of transactions. It differs in that explicit timestamp information is included in all **INFORM_COMMIT** actions. It is also similar to that of a pseudotime object (as defined in [2]) in that the object receives timestamp information for some transactions. However, hybrid objects differ from pseudotime objects in that the timestamp information is included in the **INFORM_COMMIT** actions rather than in separate **INFORM_TIME** actions; thus, timestamp information only arrives for transactions that have committed. Also, hybrid objects receive timestamp information for arbitrary transactions, not just for accesses to X.

A hybrid object automaton H_X is required to preserve *hybrid object well-formedness*, defined to include all the constraints corresponding to those in the definition of generic object well-formedness in [5]. In addition, there are restrictions on the timestamps supplied, similar to those in the definition of pseudotime object well-formedness in [2]: there is no transaction T for which there are two different timestamps, p and p', such that **INFORM_COMMIT_AT**(X)**OF**(T, p) and **INFORM_COMMIT_AT**(X)**OF**(T, p') both occur in β, and there is no timestamp p for which there are two different transactions, T and T', such that T and T' are siblings and **INFORM_COMMIT_AT**(X)**OF**(T, p) and **INFORM_COMMIT_AT**(X)**OF**(T', p) both occur in β. Notice that the same timestamp may be assigned to different transactions, so long as they are not siblings.

Hybrid Controller The hybrid controller behaves much the same as the generic controller defined in [5]. The main difference is that, when it commits a transaction, it simultaneously assigns a timestamp to that transaction; subsequently, it passes that timestamp to the hybrid objects in **INFORM_COMMIT** actions. The only constraint on the assignment of timestamps is that they get assigned to siblings in increasing order.

The assignment of timestamps is somewhat different from the assignment of pseudotimes that occurs in the pseudotime controller of [2]. In a hybrid system, individual timestamps are assigned to transactions, whereas in a distributed pseudotime system, intervals of pseudotime are assigned. Also, in a hybrid system, the timestamp for a transaction is chosen when the transaction commits, whereas in a

distributed pseudotime system, the pseudotime interval for a transaction is chosen before the transaction starts executing.

The hybrid controller we model is highly nondeterministic, in particular because each timestamp can be chosen arbitrarily, subject to the constraint that it is greater than the timestamps of all previously committed siblings. Actual implementations will restrict the nondeterminism by choosing timestamps in a controlled way. One simple method in a centralized system is to assign to each transaction the value of the clock at the instant the transaction commits. In this case, each transaction's timestamp is greater than that of *all* previously committed transactions, instead of merely the committed siblings as required. Another implementation can be obtained by assigning the timestamp i to a transaction if it is the i-th child of its parent that commits.

The hybrid controller has in its interface the actions of the transaction automata and the hybrid object automata, as well as extra actions **COMMIT** and **ABORT** for each transaction other than T_0. The code of the hybrid controller is identical to that of the generic controller from [5], except that the **COMMIT**(T) action chooses a timestamp p and records it in the state, and the **INFORM_COMMIT** action includes the appropriate timestamp.

Hybrid Systems A hybrid system is the composition of the hybrid controller, all the transaction automata (just as in the serial system), and a collection of hybrid object automata. The behaviors of a hybrid system are called hybrid behaviors. We have the following result: If β is a hybrid behavior, then for every object name X, $\beta|H_X$ is hybrid object well-formed for X.

Hybrid Atomicity Now *hybrid atomicity* is defined. The definition is almost the same as the definition of dynamic atomicity in [5] but it is based on hybrid systems instead of generic systems. It is also similar to static atomicity defined in [2], but the order used is the completion order.

Let H_X be a hybrid object automaton for object name X. Say that H_X is hybrid atomic if for all hybrid systems $\mathcal{S}$ in which H_X is associated with X, the following is true. Let β be a finite behavior of $\mathcal{S}$, $R = \mathit{completion}(\beta)$ and T a transaction name that is not an orphan in β.[4] Then $\mathit{view}(\mathit{serial}(\beta), T, R, X)$ is a serial behavior of S_X. The following theorem is an direct consequence of Theorem 3.

Theorem 1. *(Hybrid Atomicity Theorem) Let $\mathcal{S}$ be a hybrid system in which all hybrid objects are hybrid atomic. Let β be a finite behavior of $\mathcal{S}$. Then β is serially correct for every non-orphan transaction name.*

Local Hybrid Atomicity We now give a local version of hybrid atomicity. The development is analogous to that for local dynamic atomicity in Section 6 of [5] (in that we define local analogues for many concepts) but includes some significant technical changes, needed to allow us to prove that the algorithm of Section 3 is correct.

[4] Recall that a transaction T is an orphan in β if **ABORT**(U) appears in β for some ancestor U of T.

We begin by defining *local visibility* and *local-completion* exactly as in [5]. That is, if H_X is a hybrid object automaton for object name X, and β is a sequence of external actions of H_X, then T is locally visible at X to T' in β if β contains an **INFORM_COMMIT_AT**(X)**OF**(U,p) event for every U in $ancestors(T) - ancestors(T')$, and $local\text{-}completion(\beta)$ is the binary relation on accesses to X where $(U,U') \in local\text{-}completion(\beta)$ if and only if $U \neq U'$, β contains **REQUEST_COMMIT** events for both U and U', and U is locally visible at X to U' in β', where β' is the longest prefix of β not containing the given **REQUEST_COMMIT** event for U'.

In this paper we will use a different notion of *local orphans* from that in [5] and [2]. The prior definition designated a transaction T as a local orphan exactly if an **INFORM_ABORT** appears for an ancestor of T. The new definition includes additional conditions that imply that a transaction is an orphan. For example, it can be deduced that an access T' to object X is an orphan provided that T' is created and that an **INFORM_COMMIT** event occurs for an ancestor of T' without any preceding **REQUEST_COMMIT** for T'. Moreover, if such an access T' is locally visible to any transaction T, then it can also be deduced that T is an orphan.

More formally, if β is a sequence of external actions of H_X, then we define an access T' to object X to be *excluded* in β provided that β contains **CREATE**(T'), and also contains an **INFORM_COMMIT** event for an ancestor of T' with no preceding **REQUEST_COMMIT** event for T'. Then we define a transaction name T to be a local orphan in β provided that either an **INFORM_ABORT** event occurs in β for some ancestor of T, or there is some excluded access to X that is locally visible to T.

We define another binary relation, $local\text{-}timestamp(\beta)$, on accesses to X. Namely, $(T,T') \in local\text{-}timestamp(\beta)$ if and only if T and T' are distinct accesses to X, U and U' are sibling transactions that are ancestors of T and T', respectively, β contains an **INFORM_COMMIT_AT**(X)**OF**(U,p) event, and β contains an **INFORM_COMMIT_AT**(X)**OF**(U',p') event, where $p < p'$. Notice the difference between this order and the order $local\text{-}pseudotime\text{-}order(\beta)$ defined in [2], where the order was based on the timestamps of the accesses, rather than on the timestamps of the sibling ancestors of the accesses.

Before giving our definition of local hybrid atomicity, one additional technical notion is needed. Namely, define a sequence ξ of operations of X to be *transaction-respecting* provided that for every transaction name T, all the operations for descendants of T appear consecutively in ξ. Notice that if β is a hybrid behavior, T is a transaction name that is not an orphan in β, $R = completion(\beta)$, and X is an object name, then $view(\beta,T,R,X)$ is $perform(\xi)$ where ξ is transaction-respecting. Thus by only considering transaction-respecting orderings in the definition of local-views below, rather than all orderings consistent with local information, as we did in [5], we ensure that the concept of local hybrid atomicity is a closer approximation to the concept of hybrid atomicity. Thus, a wider class of correct algorithms can be verified using the definitions of this section than would have been the case if the definition of local-views did not include the restriction to transaction-respecting orderings. In particular, the algorithm that we present in Section 3 can be proved to be local hybrid atomic using the definition as given in this section.

Suppose that β is a finite hybrid object well-formed sequence of external actions of H_X and T is a transaction name. Let $local\text{-}views(\beta,T)$ be the set of sequences

defined as follows. Let Z be the set of operations occurring in β whose transactions are locally visible at X to T in β. Then the elements of *local-views*(β, T) are the sequences of the form *perform*(ξ), where ξ is a transaction-respecting total ordering of Z in an order consistent with both the partial orders *local-completion*(β) and *local-timestamp*(β) on the transaction components.

We say that hybrid object automaton H_X for object name X is locally hybrid atomic if whenever β is a finite hybrid object well-formed behavior of H_X, and T is a transaction name that is not a local orphan at X in β, then every sequence that is an element of the set *local-views*(β, T) is a finite behavior of S_X. The definitions have been chosen so that local hybrid atomicity is a sufficient condition for hybrid atomicity. The proof of this fact is analogous to that of Theorem 54 of [5] and Theorem 8 of [2]. The main new point to note is the following. In order to show that *view*$(serial(\beta), T, R, X) =$ *perform*(ξ) is an element of *local-views*$(\beta\gamma|H_X, T)$, it must be shown not only that ξ is consistent with the *local-completion* order as before, but also that it is consistent with the *local-timestamp* order and that it is transaction-respecting.

3 Dependency-Based Hybrid Locking

This section presents an algorithm, that is a natural generalization to nested transaction systems of that given in [8]. It is based on a serial dependency relation. The intuition underlying this is that two operations of a particular serial object should be related whenever the possibility of the second occurring is influenced by the presence or absence of the first. However, there are many subleties, and the precise definition that we give (taken from [2]) is chosen to be what is needed in the algorithm (both in that earlier paper and this one). We need a preliminary definition: Let R be a binary relation on operations of serial object S_X, and ξ a sequence of operations of S_X and η is a subsequence of ξ, then say that η is R-closed in ξ provided that whenever η contains an operation (T, v), it also contains all preceding operations (T', v') of ξ such that $((T', v'), (T, v)) \in R$. Now, we say that R is a serial dependency relation for S_X provided that the following holds. Whenever ξ is a finite sequence of operations of S_X (no two of which involve the same access) such that for each (T, v) in ξ there is an R-closed subsequence η of ξ where η contains (T, v) and *perform*(η) is a behavior of S_X, then *perform*(ξ) is a behavior of S_X.

The algorithm is described as a hybrid object automaton in a hybrid system. For each object name X and binary relation C between operations of X, we describe a hybrid object automaton $D_X(C)$ (a *dependency object*). In fact, a sufficient condition for $D_X(C)$ to be locally hybrid atomic is that C be a symmetric serial dependency relation.

The algorithm is closely related to the commutativity-based locking algorithm L_X of Section 8 of [5]. The main difference is that the *intentions* of concurrent transactions are not applied to the base state in the order in which **INFORM_COMMIT** events arrive, but rather in the order given by timestamps. Thus when the object learns of the commit of a subtransaction, the intentions will be transferred to the parent, but rather than being appended at the end of the parent's previous intentions, they may be inserted into the sequence in an earlier place. To reflect this

behavior in the automaton, we no longer keep the intentions list explicitly; instead, we keep a set of descendant accesses (in the state component *intset*), and keep track of the timestamps provided by the system (in the component *time*). The intentions sequence is then obtained as a derived variable whose value is computed from these components. As in commutativity-based locking, the response to an access is constrained so that the resulting operation can be performed by the serial object from a state resulting from executing the intentions sequences of the access's ancestors.

The other change from L_X is in the condition under which an access is enabled. The condition here is that there is no other access that is not locally visible to it and is related to it by C, whereas in L_X the enabling condition is that no other access that is not locally visible to it doesn't commute forward with it. The reason that we need C to be a symmetric serial dependency relation is that if an access T completes when another access T' has occurred but is not locally visible to T, then the object does not yet have sufficient information to know whether T or T' will be ordered first by the completion order. Since the return value of T is computed using only the intentions list of ancestors of T, this return value is computed without using T'; therefore, the object must be sure that even if T' commits and is serialized before T, the return value is not inappropriate. That is, the operation of T should not be affected by T'. Also, it is possible that T will be serialized before T', so the object must ensure that T does not make the previously-given response to T' inappropriate. That is, T' should not be affected by T. The definition of serial dependency relation expresses exactly this connection.

The state components of $D_X(C)$ are *s.created*, *s.commit-requested*, *s.intset*, and *s.time*. Here, *s.created* and *s.commit-requested* are sets of transactions, all initially empty. Also *s.intset* is a total function from transactions to sets of operations, initially mapping every transaction to the empty set $\emptyset$, and *s.time* is a partial function from transactions to timestamps, initially everywhere undefined.

We would like to define[5] the derived variable $total(T)$, which serves the same purpose here as in L_X, that is, it is a sequence of operations of S_X that when performed gives the effective state produced by a transaction T. We define *s.intentions*, a mapping from transaction names to sequences of operations, so that the operations in $s.intentions(T)$ are exactly those in $s.intset(T)$, and the order in which these operations occur is such that (T', v') precedes (T'', v'') if $s.time(U')$ and $s.time(U'')$ are both defined and $s.time(U') < s.time(U'')$, where U' and U'' are the sibling transactions that are ancestors of T' and T'', respectively. Now, we let $s.total(T)$ be the sequence of operations defined recursively as follows: $s.total(T_0) = s.intentions(T_0)$, and $s.total(T) = s.total(parent(T))s.intentions(T)$ for $T \neq T_0$.

The transition relation of $D_X(C)$ is as follows.

[5] the following definition is meaningful in any state that is reachable by hybrid object well-formed executions of $D_X(C)$. However, it is not meaningful in an arbitrary state.

CREATE(T)
Effect:
$s.created = s'.created \cup \{T\}$

INFORM_COMMIT_AT(X)**OF**(T, p)
Effect:
$s.intset(T) = \emptyset$
$s.intset(parent(T))$
$= s'.intset(parent(T)) \cup s'.intset(T)$
$s.intset(U)$
$= s'.intset(U)$ for $U \neq T, parent(T)$
$s.time(T) = p$

INFORM_ABORT_AT(X)**OF**(T)
Effect:
$s.intset(U) = \emptyset$,
$U \in descendants(T)$
$s.intset(U) = s'.intset(U)$,
$U \notin descendants(T)$

REQUEST_COMMIT(T,v)
Precondition:
$T \in s'.created - s'.commit\text{-}requested$
$\nexists\ U, T', v'$ such that
$((T, v), (T', v')) \in C$,
$(T', v') \in s'.intset(U)$,
$U \notin ancestors(T)$
$perform(s'.total(T)(T, v))$
is a behavior of S_X
Effect:
$s.commit\text{-}requested =$
$s'.commit\text{-}requested \cup \{T\}$
$s.intset(T) = \{(T, v)\}$
$s.intset(U) = s'.intset(U)$ for $U \neq T$

The following result is proved just like Proposition 67 of [5].

Proposition 2. *If C is a symmetric serial dependency relation then $D_X(C)$ is locally hybrid atomic.*

From this the correctness of the algorithm follows. An immediate consequence is that if $\mathcal{S}$ is a hybrid system in which each hybrid object is of the form $D_X(C)$, where C is a symmetric serial dependency relation, then every finite behavior of $\mathcal{S}$ is serially correct for all non-orphan transaction names.

Example: Dependency-Based Locking For a FIFO Queue Object Consider a system in which an object S_X represents a FIFO queue. S_X has an associated domain of values, $\mathcal{D}$, from which the entries are taken. S_X also has an associated function $kind : accesses(X) \rightarrow \{\text{"}insert\text{"}, \text{"}delete\text{"}\}$, and an associated function $data : \{T \in accesses(X) : kind(T) = \text{"}insert\text{"}\} \rightarrow \mathcal{D}$. The set of possible return values for each access T where $kind(T) = \text{"}delete\text{"}$ is $\mathcal{D}$, while an access T where $kind(T) = \text{"}insert\text{"}$ has return value "OK". The state of S_X consists of four components: *active* (either "*nil*", or the name of an access to X), *queue* (an array of elements of $\mathcal{D}$ indexed by the positive integers), *front* (a positive integer) and *back* (another positive integer). The start state s_0 has $s_0.active = \text{"}nil\text{"}$, $s_0.back = 1$, and $s_0.front = 1$ ($s_0.queue$ may be arbitrary). The transition relation is as follows:

CREATE(T)
Effect:
$s.active = T$

REQUEST_COMMIT(T,v),
for $kind(T)$ = "*insert*"
Precondition:
$s'.active = T$
v = "OK"
Effect:
$s.active$ = "*nil*"
$s.queue[s'.back] = data(T)$
$s.back = s'.back + 1$

REQUEST_COMMIT(T,v),
for $kind(T)$ = "*delete*"
Precondition:
$s'.active = T$
$s'.back > s'.front$
$s'.queue[s'.front] = v$
Effect:
$s.active$ = "*nil*"
$s.front = s'.front + 1$

Notice how the delete activity is blocked if the queue is empty (indicated by the condition $s'.front = s'.back$).

When we use the set of positive integers as timestamps, we can construct the hybrid object automaton $D_X(C)$ where C contains all pairs of operations $((T, v), (T', v'))$ where $T \neq T'$ and either $kind(T)$ = "*delete*" or $kind(T')$ = "*delete*" (or both). C is in fact a symmetric, serial dependency relation, so (by the following results) $D_X(C)$ is hybrid atomic.

Suppose T_1, T_2, T_3 and T_4 are accesses to X, with $kind(T_1) = kind(T_2) = insert$, $kind(T_3) = kind(T_4) = delete$, $data(T_1) = 6$ and $data(T_2) = 3$. The following sequence β is a schedule of $D_X(C)$.

CREATE(T_1)
CREATE(T_2)
CREATE(T_3)
CREATE(T_4)
REQUEST_COMMIT(T_2,"OK")
REQUEST_COMMIT(T_1,"OK")
INFORM_COMMIT_AT(X)**OF**(T_1, 2)

Notice that this schedule involves concurrent insertions into the queue, since the response to T_1 occurs before the fate of T_2 is known. Since insert operations do not commute, β is not a schedule of the object L_X formed when the commutativity-based locking algorithm of [5] is used. This shows that the algorithm presented here allows concurrency not available to L_X.

The schedule β can leave $D_X(C)$ in state s where $s.created = \{T_1, T_2, T_3, T_4\}$, $s.commit\text{-}requested = \{T_2, T_1\}$, $s.intset(T_0) = \{(T_1, "OK")\}$, $s.intset(T_2) = \{(T_2, "OK")\}$, and $s.time(T_1) = 2$. The derived variable $s.total(T_3)$ is just the sequence of a single operation $(T_1, "OK")$.

In the state s there is no value v for which either action **REQUEST_COMMIT**(T_3,v) or **REQUEST_COMMIT**(T_4,v) is enabled, because of the operation $(T_2, "OK")$ in $s.intset(T_2)$. In essence, a delete access can't proceed at this point because the value to be returned ought to be 6 if T_2 has a timestamp after that for T_1 or if T_2 aborts, but if T_2 commits before T_1, then the delete should return the value 3.

4 Conclusion

We have defined an appropriate structure for nested transaction systems based on hybrid atomicity, in which each transaction is given a timestamp that indicates the order (relative to its siblings) of committing. We have defined hybrid atomicity and shown that it was a local atomicity property, so that if each object is separately verified to be hybrid atomic, the whole system's correctness follows. We have defined local hybrid atomicity and shown that it is a sufficient condition for hybrid atomicity, and finally we presented and verified an algorithm that generalizes one of Herlihy and Weihl in the unnested case.

There are several directions in which this work can be extended. One is to find and verify further algorithms that provide hybrid atomicity for particular datatypes. These might keep information in more compact forms, rather than as sets of operations as used in $D_X(C)$. Another is to consider the possibility that timestamps do not give exactly the order of completion, but rather another order consistent with Lamport causality between siblings. Both the modular atomic property and the algorithm should carry over to this situation.

Acknowledgements

We thank Michael Merritt for many useful comments on this material.

References

1. J. Allchin. *An Architecture for Reliable Decentralized Systems.* PhD thesis, Georgia Institute of Technology, September 1983. Available as Technical Report GIT-ICS-83/23.
2. J. Aspnes, A. Fekete, N. Lynch, M. Merritt, and W. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Proceedings of 14th International Conference on Very Large Data Bases*, pages 431–444, August 1988.
3. J. Eppinger, L. Mummert, and A. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.
4. K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976. Also published as IBM RJ1487, December 1974.
5. A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41(1):65–156, 1990.
6. A. Fekete, N. Lynch, M. Merritt, and W. Weihl. *Atomic Transactions.* Morgan-Kaufmann, 1992.
7. M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. In *Proc. 7th ACM Symposium on Principles of Database Systems*, pages 201–210, March 1988.
8. M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991.
9. H. Korth. Locking primitives in a database system. *JACM*, 30(1), January 1983.
10. B. Liskov. Distributed computing in Argus. *Communications of ACM*, 31(3):300–312, March 1988.
11. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989. Also in Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science Massachusettes Institute of Technology, November 1988.

12. J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute Technology, 1981. Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute Technology, April 1981. Also, published by MIT Press, March 1985.
13. D.P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Massachusetts Institute Technology, 1978. Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, Massachusetts Institute Technology, September 1978.
14. P. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3), August 1984.
15. W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.
16. W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.
17. W.E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, Massachusetts Institute Technology, 1984. Technical Report MIT/LCS/TR-314, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, March 1984.

A Review of Background

In this appendix, we summarize the main concepts from our earlier work that are used in this paper. Complete details can be found in [5] and [6].

All components in our systems, transactions, objects and schedulers, will be modelled by *I/O automata* [11]. An I/O automaton A has a set of *states*, some of which are designated as *initial states*. Usually a state is given as an assignment of values to a collection of named typed variables. The automaton has *actions*, divided into *input actions*, *output actions* and *internal actions*. We refer to both input and output actions as *external actions*. The input actions model actions that are triggered by the environment of the automaton, while the output actions model the actions that are triggered by the automaton itself and are potentially observable by the environment, and internal actions model changes of state that are not directly detected by the environment. An automaton has a transition relation, which is a set of triples of the form (s', π, s), where s' and s are states, and π is an action. Such a triple means that in state s', the automaton can atomically do action π and change to state s. A *behavior* of an automaton is a sequence of external actions, generated as the automaton starts from an initial state and moves from one state to another by allowed transitions (note that any tranition involving an internal action may occur without being seen in the behavior).

We describe systems as consisting of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata, also. Thus, we define a composition operation for I/O automata, to yield a new I/O automaton. If β is a sequence of actions of a system with component A, then we denote by $\beta|A$ the subsequence of β containing all the actions of A. The definitions are chosen so that if β is a behavior of the system then $\beta|A$ is a behavior of A.

Serial systems, which consist of transaction automata and serial object automata communicating with a serial scheduler automaton, are used to characterize the cor-

rectness of a transaction-processing system. Transaction automata represent code written by application programmers in a suitable programming language. Serial object automata serve as specifications for permissible behavior of data objects in the absence of concurrency. They describe the responses the objects should make to arbitrary sequences of operation invocations, assuming that later invocations wait for responses to previous invocations. The serial scheduler handles the communication among the transactions and serial objects, and thereby controls the order in which the transactions can take steps. It ensures that no two sibling transactions are active concurrently—that is, it runs each set of sibling transactions serially. The serial scheduler is also responsible for deciding if a transaction commits or aborts. The serial scheduler can permit a transaction to abort only if its parent has requested its creation, but it has not actually been created. Thus, in a serial system, all sets of sibling transactions are run serially, and in such a way that no aborted transaction ever performs any steps. We are not proposing serial systems as interesting implementations; rather, we use them exclusively as specifications for correct behavior of other, more interesting systems.

We represent the pattern of transaction nesting by a set $\mathcal{T}$ of transaction names, organized into a tree by the mapping *parent*, with T_0 as the root. The leaves of this tree are called *accesses*. The accesses are partitioned so that each element of the partition contains the accesses to a particular object. If T is a transaction name that is an access to the object name X and v is a value, we say that the pair (T, v) is an *operation* of X. The tree structure can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In any particular execution, however, only some of these transactions will actually take steps. We imagine that the tree structure is known in advance by all components of a system. The tree will, in general, be infinite and have infinite branching.

The classical transactions of concurrency control theory (without nesting) appear in our model as the children of T_0, which models the environment in which the rest of the transaction system runs. It has actions that describe the invocation and return of the classical transactions. The only transactions that actually access data are the leaves of the transaction tree. The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly.

A *serial system* is the composition of a set of I/O automata. This set contains a transaction automaton for each non-access node of the transaction tree, a serial object automaton for each object name, and a serial scheduler. The interface to each of these automata is described next.

A non-access transaction T is modelled as a *transaction automaton* A_T, an I/O automaton. The **CREATE** input action "wakes up" the transaction. Each **REQUEST_CREATE** output action is a request by T to create a particular child transaction. Each **REPORT_COMMIT** input action reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. Each **REPORT_ABORT** input action reports to T the unsuccessful completion of one of its children, without returning any other information. The **REQUEST_COMMIT** action is an announcement by T that it has finished its work, and includes a value recording the results of that work. We leave the executions of particular transaction automata largely unconstrained; the choice of which children

to create and what value to return will depend on the particular implementation.

We model the serial specification of an object X (describing its activity in the absence of concurrency and failures) by a *serial object automaton* S_X. Recall that transaction automata are associated with non-access transactions only, and that access transactions model abstract operations on shared data objects. We associate a single I/O automaton with each object name. The external actions for each object are just the **CREATE** and **REQUEST_COMMIT** actions for all the corresponding access transactions. Although we give these actions the same kinds of names as the actions of non-access transactions, it is helpful to think of the actions of access transactions in other terms also: a **CREATE** corresponds to an invocation of an operation on the object, while a **REQUEST_COMMIT** corresponds to a response by the object to an invocation. A useful notation for operation (T, v) of an object X is that *perform*(T, v) denotes **CREATE**(T) **REQUEST_COMMIT**(T,v). This definition is extended to sequences of operations.

The third kind of component in a serial system is the serial scheduler. The transactions and serial objects are allowed to be any I/O automata whose actions and behavior satisfy simple restrictions. The serial scheduler, however, is a fully specified automaton. It runs transactions according to a depth-first traversal of the transaction tree. The serial scheduler can choose nondeterministically to abort any transaction whose parent has requested its creation, as long as the transaction has not actually been created. Each child of T whose creation is requested must be either aborted or run to commitment with no siblings overlapping its execution, before T can commit. The result of a transaction can be reported to its parent at any time after the commit or abort has occurred. The **REQUEST_CREATE** and **REQUEST_COMMIT** inputs are intended to be identified with the corresponding outputs of transaction and serial object automata, and correspondingly for the **CREATE**, **REPORT_COMMIT** and **REPORT_ABORT** output actions. The **COMMIT**(T) and **ABORT**(T) output actions are called *completion* actions for T; they mark the point in time where the decision on the fate of T is irrevocable.

The discussion in this paper assumes an arbitrary but fixed serial system, with A_T as the non-access transaction automata, and S_X as the serial object automata. We use the term *serial behaviors* for the system's behaviors. We give the name *serial actions* to the external actions of the serial system.

If β is a sequence[6] of actions, T a transaction name and X an object name, we define $\beta|T$ to be the subsequence of β consisting of the following actions: **CREATE**(T), **REQUEST_CREATE**(T'), **REPORT_COMMIT**(T',v'), **REPORT_ABORT**(T'), or **REQUEST_COMMIT**(T,v), where T' is a child of T; and we define $\beta|X$ to be the subsequence of β consisting of the actions **CREATE**(T) or **REQUEST_COMMIT**(T,v) where T is an access to X. We define *serial*(β) to be the subsequence of β consisting of serial actions.

If β is a sequence of actions and T is a transaction name, we say T is an *orphan* in β if there is an **ABORT**(U) action in β for some ancestor U of T. We say that T is *live* in β if β contains a **CREATE**(T) event but does not contain a completion event for T.

[6] We make these definitions for arbitrary sequences of actions, because we will also use them for behaviors of systems other than the serial system.

We use the serial system to specify the correctness condition that we expect other, more efficient systems to satisfy. We say that a sequence β of actions is *serially correct* for transaction name T provided that there is some serial behavior γ such that $\beta|T = \gamma|T$.

We believe that serial correctness is a natural notion of correctness that corresponds precisely to the intuition of how nested transaction systems ought to behave. Serial correctness for T_0 of all behaviors of a system guarantees that the external world will encounter only situations that can arise in serial executions.

We outline a method for proving that a concurrency control algorithm guarantees serial correctness. These ideas give formal structure to the simple intuition that a behavior of the system will be serially correct so long as there is a way to order the transactions so that when the operations of each object are arranged in that order, the result is legal for the serial specification of that object's type. In this paper we use a particular choice of serialization order, in which a transaction is serialized ahead of those of its siblings that complete after it does. If β is a sequence of actions, then define $completion(\beta)$ to be the binary relation on transaction names containing (T, T') if and only if T and T' are siblings and either there are completion events for both T and T' in β and a completion event for T precedes a completion event for T', or else there is a completion event for T in β, but there is no completion event for T' in β.

We must introduce some technical definitions. First, we define when one transaction is "visible" to another. This captures a conservative approximation to the conditions under which the activity of the first can influence the second. Let β be any sequence of actions. If T and T' are transaction names, we say that T' is *visible* to T in β if there is a **COMMIT**(U) action in β for every U in $ancestors(T') - ancestors(T)$.

We say that an operation (T, v) *occurs* in a sequence β of actions if a **REQUEST_COMMIT**(T,v) action occurs in β.

Finally we can define the sequence of actions considered in the hypothesis of Theorem 3. Suppose β is a sequence of actions, T a transaction name, $R = completion(\beta)$ and X an object name. Let ξ be the sequence consisting of those operations occurring in β whose transaction components are accesses to X and that are visible to T in β, ordered so that (T', v') precedes (T'', v'') if $(U', U'') \in R$, where U' is an ancestor of T', U'' is an ancestor of T'', and U' is a sibling of U''. Define $view(\beta, T, R, X)$ to be $perform(\xi)$.

The following result expresses the fundamental proof technique we use. It is proved as Proposition 46 of [5]. The term "simple behavior" is formally defined in [5]; informally, it refers to any sequence of actions that does not violate obvious causality principles (for example, by creating a transaction that was never requested). The concept is sufficiently general that this result can be applied to all behaviors of the hybrid system considered in this paper.

Theorem 3. *Let β be a finite simple behavior, T a transaction name such that T is not an orphan in β, and let $R = completion(\beta)$. Suppose that for each object name X, $view(\beta, T, R, X)$ is a finite behavior of S_X. Then β is serially correct for T.*

This article was processed using the LaTeX macro package with LLNCS style