ELSEVIER

Theoretical Computer Science 149 (1995) 151–178

Theoretical Computer Science

Hybrid atomicity for nested transactions

Alan Fekete [a,1], Nancy Lynch [b,*,2], William E. Weihl [b,3]

[a] *University of Sydney, Sydney, Australia*

[b] *MIT Laboratory for Computer Science, 545 Technology Square Cambridge, MA 02139, USA*

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. The results in this paper extend earlier work using the same model for locking and timestamp-based algorithms, providing further evidence for the generality of the approach. However, there are some subtle differences with the definitions used in earlier work, showing the difficulties of developing precise general models for nested transaction systems.

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 [8, 13, 16, 15, 14]). 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 non-commuting operations must conflict can hurt performance by restricting concurrency. Recently, Herlihy and Weihl proposed a new technique, based on assigning timestamps

* Corresponding author.

[1] A preliminary version of this paper appeared in the Proceedings of the International Conference on Database Theory, 1992.

[2] The second author was supported in part by the National Science Foundation under Grant CCR-86-11442, in part by the Defence 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 N00014-91-1046.

[3] The third author was supported in part by the National Science Foundation under Grant CCR-8716884, and in part by the Defence Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988. He was also supported in part by an equipment grant from Digital Equipment Corporation.

SSDI 0304-3975(95)00029-1

to transactions as they commit and propagating the timestamp information to objects, that allows some of the conflicts imposed by commutativity to be eliminated [6, 7]. 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., [12, 11, 9, 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, together with a rigorous correctness proof. 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.[4] (That is, 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, and prove that it ensures hybrid atomicity.

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, and we verify this. Finally, we conclude with a discussion and some suggestions for further work. In an Appendix, we summarize the earlier work of ours that provides the framework for this paper.

2. Hybrid atomicity

The development in this section closely parallels those in [5] and [2]. In our presentation we concentrate on those aspects that are different from the previous papers.

2.1. Hybrid systems

This section defines 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*. The transaction automata represent user-written code; they are just the same as in a

[4] Weihl defined several local properties for single-level transaction systems [16, 15]; the local property defined here generalizes one of those to nested transaction systems.

serial system. (See the appendix for a description of serial systems and the definition of correctness based on them.) Each hybrid object automaton encapsulates both data and concurrency control information that is maintained for one object; this might include multiple versions, locks and/or log entries. One can think of this automaton as a resource manager. The hybrid controller represents the transaction processing monitor; it acts as a communication medium, passing requests between the other components. As well, the hybrid controller is responsible for making the decision to commit or abort each transaction, and it generates the timestamps that are provided to the concurrency control algorithms.

Throughout, we use a totally ordered set $\mathscr{P}$ of *timestamps*. In our development, we will not actually need the set $\mathscr{P}$ to be totally ordered–it will be enough that the timestamps assigned to sibling transactions be ordered with respect to each other. However, for simplicity we assume the total ordering. A natural choice for $\mathscr{P}$ is the set of positive integers, or more realistically, the integers less than some (extremely large) maximum.

2.1.1. 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.

Input:

CREATE(T), *for T an access to X*

INFORM_COMMIT_AT(X)OF(T,p), *for $T \neq T_0$, $p \in \mathscr{P}$*

INFORM_ABORT_AT(X)OF(T), *for $T \neq T_0$*

Output:

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 has input actions to model the receipt of a request[5] for data access (CREATE), and receipt of information about the completion of transactions (INFORM_COMMIT and INFORM_ABORT). There are output actions to model the resource sending a response for a requested data access (REQUEST_COMMIT). The interface of a hybrid object automaton H_X is similar to that of a generic object G_X, as defined in [5]. 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 arrives only for transactions that have committed. Also, hybrid objects receive timestamp information for arbitrary transactions, not just for accesses to X.

[5] We assume that the transaction name encodes all relevant information about the access such as its position in the transaction tree or any arguments that must be passed to the code (e.g. the new value for an update).

Definition (*Hybrid object well-formedness*). A hybrid object automaton H_X is required to preserve *hybrid object well-formedness*, defined as follows. A sequence β of actions of H_X is said to be hybrid object well-formed for X provided that the following conditions hold.

(1) There is at most one CREATE(T) event in β for any access T.

(2) There is at most one REQUEST_COMMIT event in β for any access T.

(3) If there is a REQUEST_COMMIT event for access T in β, then there is a preceding CREATE(T) event in β.

(4) 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.

(5) 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.

(6) There is no transaction T for which both an INFORM_COMMIT event and an INFORM_ABORT event at X for T occur in β.

(7) If an INFORM_COMMIT event occurs at X for T in β and T is an access to X, then there is a preceding REQUEST_COMMIT event for T in β.

This definition reflects some obvious properties one would expect for the activity of a resource manager in a complete system. The definition includes 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]. However, notice that the same timestamp may be assigned to different transactions, as long as they are not siblings.

2.1.2. Hybrid controller

There is a single hybrid controller for each system type that acts as a communication medium. For example, it receives requests for subtransaction activity (REQUEST_CREATE) from transaction automata, and later, the activity begins (CREATE) at another transaction or at an object. 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 automaton given below 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 ith child of its parent that commits.

The hybrid controller has the following actions, which define its interface to the transactions and hybrid objects.

Input:

REQUEST_CREATE(T), $T \neq T_0$

REQUEST_COMMIT(T,v), v a value for T

Output:

CREATE(T)

COMMIT(T), $T \neq T_0$

ABORT(T), $T \neq T_0$

REPORT_COMMIT(T,v), $T \neq T_0$

REPORT_ABORT(T), $T \neq T_0$

INFORM_COMMIT_AT(X)OF(T,p), $T \neq T_0$, $p \in \mathscr{P}$

INFORM_ABORT_AT(X)OF(T), $T \neq T_0$

Each state s of the hybrid controller consists of the following components: *s.create_requested*, *s.created*, *s.commit_requested*, *s.committed*, *s.aborted* and *s.reported*, and *s.time*. The set *s.commit_requested* is a set of operations ((transaction, value) pairs), *s.time* is a partial function from $\mathscr{T}$ to $\mathscr{P}$, and the others are sets of transactions. All are empty in the start state except for *create_requested*, which is $\{T_0\}$. The first six components are the same as in the generic controller of [5]. As a convenience, we write $s.completed = s.committed \cup s.aborted$. The transitions of the hybrid controller are as follows.

REQUEST_CREATE(T)
  Effect:
    $s.create\_requested = s'.create\_requested \cup \{T\}$

REQUEST_COMMIT(T,v)
  Effect:
    $s.commit\_requested = s'.commit\_requested \cup \{(T,v)\}$

CREATE(T)
  Precondition:
    $T \in s'.create\_requested - s'.created$
  Effect:
    $s.created = s'.created \cup \{T\}$

REPORT_COMMIT(T,v)
  Precondition:
    $T \in s'.committed$
    $(T,v) \in s'.commit\_requested$
    $T \notin s'.reported$
  Effect:
    $s.reported = s'.reported \cup \{T\}$

REPORT_ABORT(T)
  Precondition:
    $T \in s'.aborted$
    $T \notin s'.reported$
  Effect:
    $s.reported = s'.reported \cup \{T\}$

COMMIT(T)
Precondition:
$(T, v) \in s'.commit\_requested$ for some v
$T \notin s'.completed$
$p > s'.time(T')$
for every $T' \in siblings(T) \cap s'.committed$
Effect:
$s.committed = s'.committed \cup \{T\}$
$s.time(T) = p$

ABORT(T)
Precondition:
$T \in s'.create\_requested - s'.completed$
Effect:
$s.aborted = s'.aborted \cup \{T\}$

INFORM_COMMIT_AT(X)OF(T, p)
Precondition:
$T \in s'.committed$
$s'.time(T) = p$

INFORM_ABORT_AT(X)OF(T)
Precondition:
$T \in s'.aborted$

Notice that these are identical to those of the generic controller from [5] except that the COMMIT(T) action chooses a timestamp p and records it as $s.time(T)$, and the INFORM_COMMIT action includes the appropriate timestamp.

2.1.3. *Hybrid systems*

A hybrid system of a given system type is the composition of a strongly compatible set of automata. For each nonaccess transaction name T there is a transaction automaton A_T for T, the same automaton as in the serial system. For each object name X there is a hybrid object automaton H_X for X. Finally, there is the hybrid controller automaton for the system type. The external actions of a hybrid system are called hybrid actions, and the executions, schedules and behaviors of a hybrid system are called hybrid executions, hybrid schedules and hybrid behaviors, respectively. We have the following result: If β is a hybrid behavior, then for every transaction name T, the projection $\beta|T$ is transaction well-formed for T, and for every object name X, the projection $\beta|H_X$ is hybrid object well-formed for X. (This follows from the definition of the hybrid controller.)

2.2. *Hybrid atomicity*

Now *hybrid atomicity* is defined. Informally, this is a property of an automaton that says that no matter what hybrid system the automaton is placed in, the responses it gives to accesses are appropriate for a serialization order that is the order in which siblings complete. 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* for a given system type if for all hybrid systems $\mathscr{S}$ of the given type in which H_X is used as the hybrid object automaton for object name X, the following is true. Let β be a finite behavior of $\mathscr{S}$, $R = completion(\beta)$ and T a transaction name that is not an orphan[6] in β. Then $view(serial(\beta), T, R, X)$ is a serial behavior of S_X.[7] Informally,

[6] A transaction T is an orphan in β if ABORT(U) appears in β for some ancestor U of T.
[7] See the appendix for definitions of the symbols used here.

this says that H_X ensures that the operations done by the transactions visible to T is consistent with serializing them in the order in which they commit, regardless of the system of which H_X is a part.

The following theorem is an direct consequence of Theorem A.1 in the appendix; it says that hybrid atomicity is a sufficient *local* condition to guarantee *global* correctness.

Theorem 2.1 (Hybrid Atomicity Theorem). *Let $\mathscr{S}$ be a hybrid system in which all hybrid objects are hybrid atomic. Let β be a finite behavior of $\mathscr{S}$. Then β is serially correct for every non-orphan transaction name.*

2.3. Local hybrid atomicity

We now give a local version of hybrid atomicity. This is similar to the definition above (and it will be shown to be a sufficient condition for hybrid atomicity), but it deals only with well-formed behaviors of the automaton itself, rather than considering all possible systems in which the automaton can be placed. The development is analogous to that for local dynamic atomicity in [5], but includes some significant technical changes, needed to allow us to prove that the algorithm of Section 3 is correct.

Definition (*Local visibility*). We need to have local forms of the concepts used to define hybrid atomicity, such as orphan transactions, visibility, and the completion order. In each case, we try to capture much of the original concept while referring only to the behavior of H_X itself. 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 $ancestors(T) - ancestors(T')$; $local\text{-}completion(\beta)$ 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'.

Definition (*Local orphan*). In this paper we will use a different notion of *local orphans* from that in [5, 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.

The relationship between these local definitions (applied to a projection) and their corresponding global versions is expressed in the following result.

Lemma 2.2. *Let β be a behavior of a hybrid system that contains hybrid object automaton H_X. If T is locally visible at X to T' in $\beta|H_X$ then T is visible to T' in β. Similarly, if T is a local orphan at X in $\beta|H_X$ then T is an orphan in β.*

The following lemma follows easily by considering the two cases of the definition of a local orphan.

Lemma 2.3. *Let H_X be a hybrid object automaton for X. Let β be a hybrid object well-formed sequence of external actions of H_X. If T' is a local orphan in β and T' is locally visible to T in β, then T is a local orphan in β.*

Definition (*Local-timestamp order*). We define another binary relation, *local-timestamp* (β), on accesses to X. Namely,$(T, T') \in$*local-timestamp*(β) 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 $p < p'$. This is the crucial aspect of this work, since the local-completion order only reflects facts about the completion order that can be deduced from the order in which accesses occur. In contrast, the local-timestamp order captures facts about the completion order that are deducible from the timestamps assigned when a transaction commits, and passed to H_X in the INFORM_COMMIT action. Notice the difference between this order and the order *local-pseudotime-order*(β) 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. The correctness of these deductions is expressed in the following lemma.

Lemma 2.4. *Let β be a behavior of a hybrid system in which hybrid object automaton H_X is associated with X, and let $R =$ completion(β). Let T and T' be accesses to X. If $(T, T') \in$ local-completion($\beta|H_X$) and T' is not an orphan in β, then $(T, T') \in R_{\text{trans}}$. If $(T, T') \in$ local-timestamp($\beta|H_X$), then $(T, T') \in R_{\text{trans}}$.*

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 a sequence of operations is totally ordered by R_{trans} for any sibling order R, then the sequence is transaction-respecting. In particular, if β is a hybrid behavior, T is a transaction name that is not an orphan in β, $R =$ *completion*(β), and X is an object name, then *view*(β, T, R, X) is *perform*(ξ) where ξ

is transaction-respecting, since it is totally ordered by R_{trans}. 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 locally 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-views*(β, 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 main result of this section, Theorem 2.5, says that local hybrid atomicity is a sufficient condition for hybrid atomicity.

Theorem 2.5. *If H_X is a hybrid object automaton for object name X that is locally hybrid atomic then H_X is hybrid atomic.*

Proof. Let $\mathcal{S}$ be a hybrid system in which H_X occurs. Let β be a finite behavior of $\mathcal{S}$, $R = \textit{completion}(\beta)$ and T a transaction name that is not an orphan in β. The proof must establish that *view*$(\textit{serial}(\beta), T, R, X)$ is a behavior of S_X. By definition, *view*$(\textit{serial}(\beta), T, R, X) = \textit{perform}(\xi)$, where ξ is the sequence of operations occurring in β whose transactions are visible to T in β, arranged in the order given by R_{trans} on the transaction components.

Let γ be a finite sequence of actions consisting of exactly one INFORM_COMMIT_AT(X)OF(U) for each COMMIT(U) that occurs in β. Then $\beta\gamma$ is a behavior of the system $\mathcal{S}$, since each action in γ is an enabled output action of the hybrid controller. Then $\beta\gamma|H_X$ is a behavior of H_X, and it is hybrid object well-formed.

Since INFORM_COMMIT_AT(X)OF(U) occurs in $\beta\gamma|H_X$ if and only if COMMIT (U) occurs in β, an access T' to X is visible to T in β if and only if it is locally visible at X to T in $\beta\gamma|H_X$. Therefore, the same operations occur in *view*$(\textit{serial}(\beta), T, R, X)$ and in any sequence in *local-views*$(\beta\gamma|H_X, T)$. To show that *view*$(\textit{serial}(\beta), T, R, X) \in$ *local* $-$ *views*$(\beta\gamma|H_X, T)$, the proof must show that the order of operations in the first sequence is among the orders considered in the latter set of sequences.

If T' is any access that is locally visible at X to T in $\beta\gamma|H_X$, then T' is visible to T in β, so T' is not an orphan in β, and hence not an orphan in $\beta\gamma$. Also, note that *completion*$(\beta\gamma)$ = *completion*$(\beta) = R$. Then Lemma 2.4 implies that if accesses that are locally visible at X to T in $\beta\gamma|H_X$ are ordered by *local-completion*$(\beta\gamma|H_X)$ or by *local-timestamp*$(\beta\gamma|H_X)$ then they are also ordered in the same way by R_{trans}.

Thus, the sequence ξ can be obtained by taking those operations (T', v') such that REQUEST_COMMIT(T', v') occurs in $\beta\gamma|H_X$ and T' is locally visible at X to T in $\beta\gamma|H_X$, and arranging them in an order that is consistent with *local-completion*$(\beta\gamma|H_X)$ and *local-timestamp*$(\beta\gamma|H_X)$ on the transaction component. The fact that ξ is transaction-respecting follows from the definition of *view*, which arranges operations in an order induced by a sibling order. Thus, *perform*(ξ) is an element of *local-views*$(\beta\gamma|H_X, T)$. Since H_X is locally hybrid atomic, *perform*(ξ) is a behavior of S_X, as required. □

3. Dependency-based hybrid locking

This section presents an algorithm that is a natural generalization to nested transaction systems of that given in [7]. The essential idea of this algorithm is to maintain information about each transaction's *intentions*. These are the accesses that were carried out by descendendants of the transaction such that every ancestor from the access up to (but not including) the transaction itself has committed. A similar concept is used in the commutativity-based locking algorithm L_X of [5]. The main difference is that in this paper, 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 do not 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 (the conflict relation), 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.

Definition (*Serial dependency relation*). The correctness of this algorithm depends on a sensible choice of C. The precise condition used is expressed in terms of the concept

of 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: if R is a binary relation on operations of serial object S_X, ξ is 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.

For correctness, we require C to be a symmetric serial dependency relation. Symmetry is needed for the following reason: 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 appropriate. 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.

Since the set of operations that do not commute forms a symmetric serial dependency relation, we see that any concurrency available using commutativity-based locking is also available with dependency-based locking. In Example 3.1, we show that dependency-based locking can allow concurrency not available to commutativity-based locking. However, this added concurrency is only obtained by requiring more of the hybrid controller than of the generic controller used in [5]. Note that the hybrid controller must determine the relative completion order between siblings, and pass this information to the objects.

3.1. The objects $D_X(C)$

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*). A sufficient condition for $D_X(C)$ to be locally hybrid atomic is that C be a symmetric serial dependency relation.

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.

Definition (*total*). We would like to define the derived variable $total(T)$, which is a sequence of operations of S_X which when performed gives the effective state produced by a transaction T. The cleanest presentation of this variable is by an expression that is not meaningful in arbitrary states of $D_X(C)$; however it is meaningful in any state that is reachable by hybrid object well-formed executions of $D_X(C)$. We identify three properties of these states.[8] First, there is at most one operation for each access transaction name in the union of all the intsets. Second, if $(T', v') \in s.intset(T)$, then T' is a descendant of T, and $s.time(T'')$ is defined for all $T'' \in s.ancestors(T') - s.ancestors(T)$. Third, whenever U and U' are siblings such that $s.time(U)$ and $s.time(U')$ are both defined, then $s.time(U) \neq s.time(U')$.

In any state s that satisfies the properties above, we can use the value of the variable $s.time$ to define a binary relation R on accesses to X, as follows. If T and T' are distinct accesses to X, then we define $(T, T') \in R$ exactly 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. Note that if β is a hybrid object well-formed schedule of $D_X(C)$ that can lead to state s, then R coincides with the relation $local\text{-}timestamp(\beta)$.

The three properties above ensure that R is a partial order that totally orders the operations in $s.intset(T)$, for any transaction name T. Thus in a state s that satisfies the properties, we define a derived variable $s.intentions$, which is a mapping from transaction names to sequences of operations. Namely, if T is any transaction name, then the operations in the sequence $s.intentions(T)$ are exactly those in $s.intset(T)$. The order in which these operations occur is determined by the relation R defined from $s.time$, as described just above.

Finally, 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.

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 $U \neq T.parent(T)$
    $s.time(T) = p$

REQUEST_COMMIT(T,v), T an access to X
  Precondition:
    $T \in s'.created - s'.commit\_requested$
    $\nexists\ U, T'$, and v' such that
      $((T, v), (T', v')) \in C$
      $(T', v') \in s'.intset(U)$, and
      $U \notin ancestors(T)$
    $perform(s'.total(T)(T, v))$
      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 all $U \neq T$

[8] To be precise, and to avoid circular reasoning, one should define the transition relation so that REQUEST_COMMIT is not enabled in states that do not satisfy the properties given here, while in states that do satisfy these properties it is enabled if the precondition as listed below (which is then well-defined) is true. One can then prove that each state reachable in a hybrid object well-formed execution satisfies these properties, so that in these states the transition relation is just as listed.

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

Example 3.1 (*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, $\mathscr{D}$, from which the entries are taken. S_X also has an associated function $kind : accesses(X) \to \{\text{"}insert\text{"},\text{"}delete\text{"}\}$, and an associated function $data : \{T \in accesses(X) : kind(T) = \text{"}insert\text{"}\} \to \mathscr{D}$. The set of possible return values for each access T where $kind(T) = \text{"}delete\text{"}$ is $\mathscr{D}$, 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 $\mathscr{D}$ 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 $kind(T) = \text{"}insert\text{"}$
  Precondition:
    $s'.active = T$
    $v = \text{"}OK\text{"}$
  Effect:
    $s.active = \text{"}nil\text{"}$
    $s.queue[s'.back] = data(T)$
    $s.back = s'.back + 1$

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

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 $kind(T) = \text{"}delete\text{"}$ or $kind(T') = \text{"}delete\text{"}$ (or both). C is in fact a symmetric, serial dependency relation, so (by the results of the following section) $D_X(C)$ 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 $s.created = \{T_1, T_2, T_3, T_4\}$, $s.commit\text{-}requested = \{T_2, T_1\}$, $s.intset(T_0) = \{(T_1, \text{"}OK\text{"})\}$, $s.intset(T_2) = \{(T_2, \text{"}OK\text{"})\}$, and $s.time(T_1) = 2$. The derived variable $s.total(T_3)$ is just the sequence of a single operation $(T_1, \text{"}OK\text{"})$.

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 $(T_2, \text{"}OK\text{"})$ 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 3 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 6.

3.2. Correctness proof

The correctness proof roughly follows the corresponding one in [5] for commutativity-based locking. We define lock-visible and lock-completion exactly as in that paper. That is, we say that T is *lock-visible* at X to T' in β if β contains a subsequence β' consisting of an INFORM_COMMIT_AT(X)OF(U, p_U) event for every $U \in ancestors(T) - ancestors(T')$, arranged in ascending order (so the INFORM_COMMIT for $parent(U)$ is preceded by that for U). Also we say that $(U, U') \in lock\text{-}completion(\beta)$ if and only if $U \neq U'$, β contains REQUEST_COMMIT events for both U and U', and U is lock-visible to U' at X in β', where β' is the longest prefix of β not containing the given REQUEST_COMMIT event for U'.

We have a basic lemma relating lock-visibility and lock-completion to the corresponding local properties.

Lemma 3.1. *Let β be a sequence of actions of $D_X(C)$ and T and T' transaction names. If T is lock-visible at X to T' in β then T is locally visible at X to T' in β. Also lock-completion(β) is a subrelation of local-completion(β).*

The following lemmas relate the intsets to lock-visibility and lock-completion.

Lemma 3.2. *Let β be a finite hybrid object well-formed behavior of $D_X(C)$. Suppose that β can leave $D_X(C)$ in state s. If $(T, v) \in s.intset(T')$ then T is a descendant of T',* REQUEST_COMMIT(T, v) *occurs in β, and T' is the highest ancestor of T to which T is lock-visible at X in β.*

Lemma 3.3. *Let β be a finite hybrid object well-formed behavior of $D_X(C)$, and suppose that a* REQUEST_COMMIT(T, v) *event π occurs in β, where T is not a*

local orphan at X in β. Let β' be the prefix of β ending with π, and let s be the (unique) state in which β' can leave $D_X(C)$. Then the following are true.

(1) *The operations in $s.total(T)$ are exactly (T,v) plus the operations (T',v') that occur in β such that $(T',T) \in$ lock-completion(β).*

(2) *perform($s.total(T)$) is a finite behavior of S_X.*

The key lemma is next, which shows that certain sequences of actions, extracted from a hybrid object well-formed behavior of $D_X(C)$, are serial object well-formed behaviors of S_X. This lemma is somewhat similar to Lemma 66 of [5], but the proof is changed in significant ways. As in that lemma, we need an extra definition. Suppose β is a hybrid object well-formed finite behavior of $D_X(C)$. Then a set Z of operations of X is said to be *allowable* for β provided that for each operation (T,v) that occurs in Z, the following conditions hold.

(1) (T,v) occurs in β.

(2) T is not a local orphan at X in β.

(3) If (T',v') is an operation that occurs in β such that $(T',T) \in$ *lock-completion* (β), then $(T',v') \in Z$.

Notice that this definition uses the same words as the one in [5]; however, recall that in this case the term *local orphan* has a new (extended) meaning.

Lemma 3.4. *Suppose that C is a symmetric serial dependency relation. Let β be a finite hybrid object well-formed behavior of $D_X(C)$ and let Z be an allowable set of operations for β. If ξ is a transaction-respecting total ordering of Z that is consistent with both lock-completion(β) and local-timestamp(β) on the transaction components, then perform(ξ) is a finite behavior of S_X.*

Proof. We use the definition of a serial dependency relation. Since no two operations in Z have the same transaction name, ξ is a serial object well-formed sequence of operations of S_X. Let (T,v) be any operation in ξ. We will produce a C-closed subsequence η of ξ containing (T,v) such that *perform*(η) is a behavior of S_X. Since C is a serial dependency relation, it will follow that *perform*(ξ) is a finite behavior of S_X, as needed.

Let β_1 be the prefix of β strictly preceding REQUEST_COMMIT(T,v) and let s_1 be the state resulting from β_1; also let β_2 be β_1REQUEST_COMMIT(T,v) and let s_2 be the state resulting from β_2. Let η be the sequence $s_2.total(T)$, and let Z' be the set of operations occurring in η. By Lemma 3.3, Z' consists exactly of (T,v), together with the set of operations (T',v') occurring in β such that $(T',T) \in$ *lock-completion*(β).

The precondition and effect of the REQUEST_COMMIT(T,v) action immediately imply that *perform*(η) is a finite behavior of S_X.

We show that η is a subsequence of ξ. First, we show that $Z' \subseteq Z$. As noted above, Z' consists exactly of (T,v), together with the set of operations (T',v') occurring in β such that $(T',T) \in$ *lock-completion*(β). By assumption, $(T,v) \in Z$. If $(T',v') \in Z'$

and $(T', T) \in$ *lock-completion*(β), then since $(T, v) \in Z$ and Z is allowable, it follows that $(T', v') \in Z$. Therefore, $Z' \subseteq Z$.

Now we show that the order of operations in η is the same as that in ξ. Suppose (T_1, v_1) and (T_2, v_2) are two distinct operations in η, where (T_1, v_1) precedes (T_2, v_2) in η. We show that (T_1, v_1) precedes (T_2, v_2) in ξ. There are two cases to consider. (These are exhaustive.)

(1) (T_1, v_1) precedes (T_2, v_2) in s_2.*intentions*(U) for the same transaction U. Then none of T_1, T_2 or U is equal to T, since the effect of the REQUEST_COMMIT event is to place an operation of T alone in *intset*(T). Since s_2.*intset*$(U) = s_1$.*intset*(U) for $U \neq T$, we see that (T_1, v_1) precedes (T_2, v_2) in s_1.*intentions*(U), and Lemma 3.2 implies that T_1 and T_2 are both descendants of U and are *lock-visible* to U in β_1. Then both are locally visible to U in β_1, by Lemma 3.1. Let U_1 and U_2 be the children of *lca*(T_1, T_2) that are ancestors of T_1 and T_2, respectively. By definition of local visibility, it must be that INFORM_COMMIT events occur in β_1 for both U_1 and U_2. Since (T_1, v_1) precedes (T_2, v_2) in s_1.*intentions*(U), it must be that s_1.*time*$(U_1) <$ s_1.*time*(U_2). Since *time* records the timestamps from the INFORM_COMMIT events in β, it follows that $(T_1, T_2) \in$ *local-timestamp*(β). Since ξ is consistent with *local-timestamp*(β), it follows that (T_1, v_1) precedes (T_2, v_2) in ξ.

(2) $(T_1, v_1) \in s_2$.*intset*(U) and $(T_2, v_2) \in s_2$.*intset*(V), for some transactions U and V, where U is a proper ancestor of V.

Then let U' be the ancestor of V that is a child of U. Then T and T_2 are both descendants of V, and hence of U'. However, we claim that T_1 is not a descendant of U'. Suppose it is; then, there must be an INFORM_COMMIT event for U' in β_2, since (as T_1 is lock-visible in β_2 to U by Lemma 3.2) there is an INFORM_COMMIT event in β_2 for each ancestor of T_1 that is a proper descendant of U. As $\beta_2 = \beta_1$ REQUEST_COMMIT(T, v), we deduce that there is an INFORM_COMMIT event for U' in β_1, where U' is an ancestor of T. That is, T is an excluded access in β, which implies that T is a local orphan in β, by the new definition of this paper. This contradicts the fact that Z is an allowable set. This contradiction establishes the claim that T_1 is not a descendant of U'.

Also, $(T_1, T) \in$ *lock-completion*(β), by the characterization of Z' given above. (Note that (T_1, v_1) is not (T, v), because (T_1, v_1) appears in s_2.*intset*(U) and U is not an access, as it has a proper descendant.) Since ξ is consistent with *lock-completion*(β), it must be that (T_1, v_1) precedes (T, v) in β. Since ξ is transaction-respecting, it must also be that (T_1, v_1) precedes (T_2, v_2) in ξ.

Thus, in either case, (T_1, v_1) precedes (T_2, v_2) in ξ, which implies that η is a subsequence of ξ. Now we show that η is C-closed in ξ. Suppose that (T_1, v_1) appears in η, (T_2, v_2) precedes (T_1, v_1) in ξ, and $((T_2, v_2), (T_1, v_1)) \in C$. We must show that (T_2, v_2) appears in η. There are two cases.

(1) REQUEST_COMMIT(T_1, v_1) *precedes* REQUEST_COMMIT(T_2, v_2) *in* β: Then let β_3 be the prefix of β ending with REQUEST_COMMIT(T_2, v_2), and s_3 the state after β_3. Then since $((T_2, v_2), (T_1, v_1)) \in C$, the definition of REQUEST_COMMIT(T_2, v_2) implies that $(T_1, v_1) \in s_3$.*intset*(U) for some ancestor U of T_2. Therefore, by Lemma

3.3 and the fact that T_2 is not a local orphan since Z is allowable, $(T_1, T_2) \in$ *lock-completion*(β). Since ξ is consistent with *lock-completion*(β), it follows that (T_1, v_1) precedes (T_2, v_2) in ξ. This is a contradiction.

(2) REQUEST_COMMIT(T_2, v_2) *precedes* REQUEST_COMMIT(T_1, v_1) *in* β: Then let β_3 be the prefix of β ending with REQUEST_COMMIT(T_1, v_1), and s_3 the state after β_3. Then since $((T_2, v_2), (T_1, v_1)) \in C$ and C is symmetric, the definition of REQUEST_COMMIT(T_1, v_1) implies that $(T_2, v_2) \in s_3.intset(U)$ for some ancestor U of T_1. Then by Lemma 3.3 and the fact that T_1 is not a local orphan, $(T_2, T_1) \in$ *lock-completion*(β).

Since (T_1, v_1) is in η, either $(T_1, v_1) = (T, v)$ or $(T_1, T) \in$ *lock-completion*(β), by the characterization of Z'. Then transitivity of *lock-completion*(β) implies that $(T_2, T) \in$ *lock-completion*(β). The characterization of Z' then implies that (T_2, v_2) appears in η.

Thus, we have shown that (T_2, v_2) appears in η, which implies that η is C-closed in ξ. The definition of a serial dependency relation then implies that *perform*(ξ) is a finite behavior of S_X, as needed to complete the proof of the lemma. □

Using the previous lemma, the following result is proved just like Proposition 67 of [5].

Proposition 3.5. *If C is a symmetric serial dependency relation then $D_X(C)$ is locally hybrid atomic.*

From this the correctness of the algorithm follows.

Theorem 3.6. *If C is a symmetric serial dependency relation, then $D_X(C)$ is hybrid atomic.*

An immediate consequence of 3.6 and 2.1 is that if $\mathscr{S}$ 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 $\mathscr{S}$ is serially correct for all non-orphan transaction names.

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.

Appendix 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, 2]. The reader who is already familiar with our work, or who is not interested in the details of the proofs, may skip or skim this section.

A.1. The input/output automaton model

The following is a brief introduction to the formal model that we use to describe and reason about systems. This model is treated in detail in [10].

Definition (*States, actions*). All components in our systems, transactions, objects and schedulers, will be modelled by *I/O automata*. 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*. 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. This triple means that in state s', the automaton can atomically do action π and change to state s. An element of the transition relation is called a *step* of the automaton.

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.

Given a state s' and an action π, we say that π is *enabled* in s' if there is a state s for which (s', π, s) is a step. We require that each input action π be enabled in each state s', i.e., that an I/O automaton must be prepared to receive any input action at any time.

A *finite execution* of A is a finite alternating sequence $s_0\pi_1 s_1 \pi_2 \ldots \pi_n s_n$ of states and actions of A, ending with a state, such that s_0 is a start state and each triple (s', π, s) that occurs as a consecutive subsequence is a step of A.

Definition (*behavior*). From any finite execution, we can extract the *behavior*, which is the subsequence consisting of the external actions of A. This represents the information that the environment can detect about the execution. Since the same action may occur

several times in an execution or behavior, we refer to a single occurrence of an action as an *event*.

We say that a behavior β *can leave* A *in* state s if there is some execution with behavior β and final state s.

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. A collection of I/O automata is said to be *strongly compatible* if any internal action of any one automaton is not an action of any other automaton in the collection, any output action of one is not an output action of any other, and no action is shared by infinitely many automata in the collection. A collection of strongly compatible automata may be composed to create a *system* $\mathscr{S}$.

A state of the composed automaton is a tuple of states, one for each component automaton, and the start states are tuples consisting of start states of the components. An action of the composed automaton is an action of a subset of the component automata. It is an output of the system if it is an output for any component. It is an internal action of the system if it is an internal action of any component. During an action π of $\mathscr{S}$, each of the components that has action π carries out the action, while the remainder stay in the same state. 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. Clearly, if β is a finite behavior of the system then $\beta|A$ is a finite behavior of A.

Let A and B be automata with the same external actions. Then A is said to *implement* B if every finite behavior of A is a finite behavior of B. One way in which this notion can be used is the following. Suppose we can show that an automaton A is "correct," in the sense that its finite behaviors all satisfy some specified property. Then if another automaton B implements A, B is also correct.

A.2. Serial systems and correctness

In this section of the paper we summarize the definitions for serial systems, which consist of transaction automata and serial object automata communicating with a serial scheduler automaton. Serial systems are used to characterize the correctness 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 absense 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.

A serial system would not be an interesting transaction-processing system to implement. It allows no concurrency among sibling transactions, and has only a very limited ability to cope with transaction failures. However, 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, a *system type*, by a set $\mathscr{T}$ of transaction names, organized into a tree by the mapping *parent*, with T_0 as the root. In referring to this tree, we use traditional terminology, such as *child*, *leaf*, *ancestor*, *lca* (that is, least common ancestor), and *descendant*. (A transaction is its own ancestor and descendant.) 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. In addition, the system type specifies a set of *return values* for transactions (henceforth simply called *values*). 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 a "mythical" transaction, T_0, the root of the transaction tree. Transaction T_0 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. It is often natural to reason about T_0 in the same way as about all of the other transactions. The only transactions that actually access data are the leaves of the transaction tree, and thus they are distinguished as "accesses." (Note that leaves may exist at any level of the tree below the root.) 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 of a given system type 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. These automata are described below.

A.2.1. Transactions

A non-access transaction T is modelled as a *transaction automaton* A_T, an I/O automaton with the following external actions. (In addition, A_T may have arbitrary internal actions.)

Input:
CREATE(T)
REPORT_COMMIT(T', v'), for every child T' of T, and every return value v' for T'
REPORT_ABORT(T'), for every child T' of T
Output:
REQUEST_CREATE(T'), for every child T' of T
REQUEST_COMMIT(T, v), for every return value v for T.

The CREATE input action "wakes up" the transaction. The REQUEST_CREATE output action is a request by T to create a particular child transaction. The 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. The 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. For the purposes of the systems studied here, the transactions are "black boxes." Nevertheless, it is convenient to assume that behaviors of transaction automata obey certain syntactic constraints, for example that they do not request the creation of children before they have been created themselves and that they do not request to commit before receiving reports about all the children whose creation they requested. We therefore require that all transaction automata preserve *transaction well-formedness*, as defined formally in [5].

A.2.2. Serial objects

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. Thus, 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 with the following external actions. (In addition, S_X may have arbitrary internal actions.)

Input:
CREATE(T), for every access T to X
Output:
REQUEST_COMMIT(T, v), for every access T to X and every return value v for T

As with transactions, while specific objects are left largely unconstrained, it is convenient to require that behaviors of serial objects satisfy certain syntactic conditions. Let

α be a sequence of external actions of S_X. We say that α is *serial object well-formed* for X if it is a prefix of a sequence of the form

CREATE(T_1)REQUEST_COMMIT (T_1, v_1) CREATE(T_2)REQUEST_COMMIT (T_2, v_2)..

where $T_i \neq T_j$ when $i \neq j$. We require that every serial object automaton preserve serial object well-formedness.[9]

A.2.3. Serial sheduler

The third kind of component in a serial system is the serial scheduler. The transactions and serial objects have been specified to be any I/O automata whose actions and behavior satisfy simple restrictions. The serial scheduler, however, is a fully specified automaton, particular to each system type. 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 actions of the serial scheduler are as follows.

Input:

REQUEST_CREATE(T, v), $T \neq T_0$

REQUEST_COMMIT(T, v)

Output:

CREATE(T)

COMMIT(T), $T \neq T_0$

ABORT(T), $T \neq T_0$

REPORT_COMMIT(T, v),$T \neq T_0$

REPORT_ABORT(T, v), $T \neq T_0$

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 and ABORT output actions mark the point in time where the decision on the fate of the transaction is irrevocable. The details of the states and transition relation for the serial scheduler can be found in [5].

A.2.4. Serial systems and serial behaviors

A *serial system* is the composition of a strongly compatible set of automata consisting of a transaction automaton A_T for each non-access transaction name T, a serial object automaton S_X for each object name X, and the serial scheduler automaton for the given system type.

[9] This is formally defined in [5] and means that the object does not violate well-formedness unless its environment has done so first.

The discussion in the remainder of this paper assumes an arbitrary but fixed system type and 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. The COMMIT(T) and ABORT(T) actions are called *completion* actions for T.

We introduce some notation that will be useful later. Let T be any transaction name. If π is one of the serial actions COMMIT(T), REQUEST_CREATE(T'), REPORT_COMMIT($T'v'$), REPORT_ABORT(T'), or REQUEST_COMMIT(T, v), where T' is a child of T, then we define *transaction*(π) to be T. If π is any serial action, then we define *hightransaction*(π) to be *transaction*(π) if π is not a completion action, and to be T, if π is a completion action for a child of T. Also, if π is any serial action, we define *lowtransaction*(π) to be *transaction*(π) if π is not a completion action, and to be T, if π is a completion action for T. If π is a serial action of the form CREATE(T), or REQUEST_COMMIT(T, v), where T is an access to X, then we define *object*(π) to be X.

Definition (*Projection, serial*). If β is a sequence[10] of actions, T a transaction name and X an object name, we define $\beta|T$ to be the subsequence of β consisting of those serial actions π such that *transaction*(π) $= T$, and we define $\beta|X$ to be the subsequence of β consisting of those serial actions π such that *object*(π) $= X$. 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 the T is *live* in β if β contains a CREATE(T), event but does not contain a completion event for T.

A.2.5. Serial Correctness

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 will be interested primarily in showing, for particular systems of automata, representing data objects that use different methods of concurrency control and a controller that passes information between transactions and objects, that all finite behaviors are serially correct for T_0.

We believe serial correctness to be a natural notion of correctness that corresponds precisely to the intuition of how nested transaction systems ought to behave. Serial correctness for T is a condition that guarantees to implementors of T that their code will encounter only situations that can arise in serial executions. Correctness for T_0 is a special case that guarantees that the external world will encounter only situations that can arise in serial executions.

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

A.3. Simple systems and serial correctness

In this section we outline a method for proving that a concurrency control algorithm guarantees serial correctness. This method is treated in more detail in [5], and is an extension to nested transaction systems of ideas presented in [16, 15]. 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 which complete after it does.

In this paper a particular system architecture is used, but the method is quite general, and so is presented in terms of a "simple system", which embodies the common features of most transaction-processing systems, independent of their concurrency control and recovery algorithms, and even of their division into modules to handle different aspects of transaction-processing.

A.3.1. Simple systems

Many complicated transaction-processing algorithms can be understood as implementations of the simple system. For example, a system containing separate objects that manage locks and a "controller" that passes information among transactions and objects can be represented in this way.

We first define an automaton called the *simple database*. There is a single simple database for each system type. The actions of the simple database are those of the composition of the serial scheduler with the serial objects. The simple database embodies those constraints that we would expect any reasonable transaction-processing system to satisfy. It does not allow CREATE, ABORT or COMMIT events without an appropriate preceding request, does not allow any transaction to have two creation or completion events, and does not report completion events that never happened. Also, it does not produce responses to accesses that were not invoked, nor does it produce multiple responses to accesses. On the other hand, the simple database allows almost any ordering of transactions, allows concurrent execution of sibling transactions, and allows arbitrary responses to accesses. We do not claim that the simple database produces only serially correct behaviors; rather, we use the simple database to model features common to more sophisticated systems that do ensure correctness.

A *simple system* is the composition of a strongly compatible set of automata consisting of a transaction automaton A_T for each non-access transaction name T, and the simple database automaton for the given system type. When the particular simple system is understood from context, we will use the term *simple behaviors* for the system's behaviors.

The Serializability Theorem is formulated in terms of simple behaviors; it provides a sufficient condition for a simple behavior to be serially correct for a particular

transaction name T. Since the simple system is so unrestrictive, results about it can be transfered to realistic systems like the hybrid system of this paper.

A.3.2. Proving serial correctness

We must introduce some technical definitions.

The type of transaction ordering needed for our theorem is more complicated than that used in the classical theory, because of the nesting involved here. Instead of just arbitrary total orderings on transactions, we will use partial orderings that only relate siblings in the transaction nesting tree. Formally, a *sibling order* R is an irreflexive partial order on transaction names such that $(T, T') \in R$ implies $parent(T) = parent(T')$.

A sibling order R can be extended in two natural ways. First, R_{trans} is the binary relation on transaction names containing (T, T') exactly when there exist transaction names U and U' such that T and T' are descendants of U and U' respectively, and $(U, U') \in R$. Second, if β is any sequence of actions, then $R_{\text{event}}(\beta)$ is the binary relation on events in β containing (ϕ, π) exactly when ϕ and π are distinct serial events in β with low transactions T and T' respectively, where $(T, T') \in R_{\text{trans}}$. It is clear that R_{trans} and $R_{\text{event}}(\beta)$ are irreflexive partial orders.

Definition (*Completion order*). In systems using hybrid atomicity, the sibling order used for serialization is the order in which sibling transactions complete. 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 one of the following holds:

(1) There are completion events for both T and T' in β, and a completion event for T precedes a completion event for T'.

(2) There is a completion event for T in β, but there is no completion event for T' in β.

Definition (*visibility*). Next, 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 $ancestors(T') - ancestors(T)$. Thus, every ancestor of T' up to (but not necessarily including) the least common ancestor of T and T' has committed in β. If β is any sequence of actions and T is a transaction name, then $visible(\beta, T)$ denotes the subsequence of β consisting of serial actions π with $hightransaction(\pi)$ visible to T in β.

Definition (*perform*). We introduce some terms for describing sequences of operations. For any operation (T, v) of an object X, let $perform(T, v)$ denote the sequence of actions CREATE(T) REQUEST_COMMIT(T, v). This definition is extended to sequences of operations: if $\xi = \xi'(T, v)$ then $perform(\xi) = perform(\xi')\ perform(T, v)$. A sequence ξ of operations of X is *serial object well-formed* if no two operations in ξ have the same

transaction name. Thus if ξ is a serial object well-formed sequence of operations of X, then *perform*(ξ) is a serial object well-formed sequence of actions of X. We say that an operation (T,v) *occurs* in a sequence β of actions if a REQUEST_COMMIT(T,v) action occurs in β. Thus, any serial object well-formed sequence β of external actions of S_X is either *perform*(ξ) or *perform*(ξ)CREATE(T) for some access T, where ξ is a sequence consisting of the operations that occur in β.

Definition (*view*). Finally we can define the "view" of a transaction at an object, according to the completion order in a behavior. This is the fundamental sequence of actions considered in the hypothesis of the Serializability theorem. Suppose β is a finite simple behavior, 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 according to R_{trans} on the transaction components. (It is a fact that this ordering is uniquely determined.) Define $view(\beta, T, R, X)$ to be *perform*(ξ).

The following result expresses the fundamental proof technique we use. It is proved as Proposition 46 of [5].

Theorem A.1. *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.*

Acknowledgement

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

References

[1] J. Allchin, An architecture for reliable decentralized systems, Ph.D. 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: *Proc. 14th Internat. Conf. on Very Large Data Bases* (1988) 431–444.

[3] J. Eppinger, L. Mummert and A. Spector, *Camelot and Avalon: A Distributed Transaction Facility* (Morgan Kaufmann, Palo Alto, CA, 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, *Commun. ACM* **19**(11) (1976) 624–633; also published as IBM RJ1487, December 1974.

[5] A. Fekete, N. Lynch, M. Merritt and W. Weihl. Commutativity-based locking for nested transactions, *J. Comput. System Sci.* **41**(1) (1990) 65–156.

[6] M.P. Herlihy and W.E. Weihl, Hybrid concurrency control for abstract data types. in: *Proc. 7th ACM Symp. on Principles of Database Systems* (1988) 201–210.

[7] M.P. Herlihy and W.E. Weihl, Hybrid concurrency control for abstract data types, *J. Comput. System Sci.* **43**(1) (1991) 25–61.

[8] H. Korth, Locking primitives in a database system, *J. ACM* **30**(1) (1983).

[9] B. Liskov, Distributed computing in Argus, *Comm. ACM* **31**(3) (1988) 300–312.

[10] N. Lynch and M. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: *Proc. 6th ACM Symp. on Principles of Distributed Computation*, (1987) 137–151; Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA., April 1987.

[11] J.E.B. Moss, Nested transactions: an approach to reliable distributed computing, Ph.D. 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.

[12] D.P. Reed, Naming and synchronization in a decentralized computer system, Ph.D. Thesis, Massachusetts Institute Technology, 1978. Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, Massachusetts Institute Technology, September 1978.

[13] P. Schwarz and A. Z. Spector, Synchronizing shared abstract types, *ACM Trans. Comput. Systems* **2**(3) (1984).

[14] W.E. Weihl, Commutativity-based concurrency control for abstract data types, *IEEE Trans. Comput.* **37**(12) (1988) 1488–1505.

[15] W.E. Weihl, Local atomicity properties: modular concurrency control for abstract data types, *ACM Trans. Programming Languages Systems* **11**(2) (1989) 249–282.

[16] W.E. Weihl, *Specification and implementation of atomic data types*, Ph.D. Thesis, Massachusetts Institute Technology, 1984. Technical Report MIT/LCS/TR-314, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, March 1984.