DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Atomic Transactions for Multiprocessor Programming: A Formal Approach

NANCY LYNCH

May 9, 1994

1. Introduction

The *atomic transaction* abstraction originated in the setting of concurrent database processing, and has proved to be successful in the more general setting of data-oriented distributed computing. This notion allows the programmer to specify a collection of elementary data-processing operations that are to be performed "as if" atomically. The collection of operations should be executed *either in its entirety or not at all*, and the originator of the transaction should be informed as to which of these two has occurred. Moreover, the transaction should appear as if all of its actions occurred at a single moment, *without any interruption from other transactions*. This transaction semantics should hold in spite of the usual sorts of failures that occur in distributed networks, such as stopped processors and links. A useful extension of the basic notion is to allow transactions to be *nested*, that is, each transaction can have subtransactions that are themselves to be executed atomically with respect to each other.

The notion of transaction is easy to use in distributed programming, because it permits the programmer to pretend that his/her programs are being executed on a sequential processor. On the other hand, the costs of providing such a clean user interface, in terms of latency and amount of communication, can be high. These costs can often be reduced by careful consideration of exactly which operations need to be performed atomically, and by making appropriate use of information about the data types of the objects.

1991 *Mathematics Subject Classification.* 68-02, 68N15, 68P15, 68Q22, 68Q60.
This work was supported by DARPA contract N00014-92-J-4033, NSF contract 9225124-CCR, and AFOSR-ONR contract F49620-94-1-0199.

It seems to me to be likely that the concept of atomic transaction will turn out to be a useful language construct for multiprocessors as well as distributed systems. This opinion is mostly based on the fact that modern multiprocessors are becoming more and more like distributed systems: their architectures are often based on networks rather than shared memory, and they can be subject to unpredictable timing and failures. Also, many of the applications being run on multiprocessors (e.g., the human genome project) appear to have a data-oriented flavor that suggests that transactions will be useful. If so, there is a good deal of work on transaction processing in distributed systems that can be carried over to the new setting to provide a basis for similar work on multiprocessors.

In this paper, I describe a formal framework for describing and reasoning about atomic transactions, including nested transactions. I indicate how this framework is used to specify the correctness conditions that nested transactions are supposed to satisfy, to describe a wide range of implementing algorithms, and to give rigorous proofs that those algorithms are correct. This framework is summarized here from the new book "Atomic Transactions", by Lynch, Merritt, Weihl and Fekete [**15**]; this framework was originally designed for reasoning about distributed algorithms, but it is sufficiently abstract that it applies directly to multiprocessors as well.

I believe that the correctness notions for nested transactions are sufficiently subtle, and the implementing algorithms sufficiently complex, that such a formal framework is crucial to the successful construction of transaction-based multiprocessor systems. In particular, it is very likely that clever optimizations for known implementing algorithms will be developed to fit them to particular multiprocessor architectures; the formal framework can be used to give precise descriptions of those optimizations and argue why they are correct.

The framework in [**15**] is based on earlier work on the "classical theory" of database concurrency control, by Bernstein, Hadzilacos and Goodman [**2**]. That work needed to be extended, however, to allow handling of nested transactions instead of just single-level transactions, and of arbitrary data types instead of just read-write objects. These extensions are important for the construction of efficient implementations for transaction systems.

2. Formal Model

I begin by describing the formal model that is used for describing the correctness conditions to be satisfied by nested transaction systems, as well as for describing the algorithms used to guarantee these conditions. The model is presented in two steps. First, I describe an underlying automaton model upon which all this work is based, and second, I describe how to model transaction systems using the basic automaton model.

2.1. I/O automata. An important part of the philosophy followed here is that fundamental work on modelling and reasoning about systems should be carried out in terms of a clean and simple *semantic model*, specifically, a (not necessarily finite-state) *automaton* model, rather than in terms of any particular specification language, programming language or proof logic. The justification for this choice is that the systems are fundamentally mathematical objects, which can be reasoned about using any combination of the tools that mathematics offers. Defining the systems as automata allows the flexibility of using a variety of different languages and logics to describe and reason about them.

Because the algorithms to be studied are *asynchronous* (i.e., they work without any assumptions about relative speeds of the system components), the model that we use is the I/O automaton model of Lynch and Tuttle [**16, 17**]. Actually, since we only prove safety and not liveness properties, a simple special case of the model suffices. Namely, we only require an automaton to come equipped with a set of *states*, a nonempty subset of *start* states, a set of *actions*, classified as *input*, *output* or *internal* actions, and a set of *steps*. Each *step* is a triple consisting of a pre-state, an action and a post-state. The interesting behavior of an automaton is described by its set of finite *traces* (sequences of external actions), and the notion that one automaton implements another is captured by inclusion of the sets of traces. The model has a collection of useful operations defined, most importantly, parallel composition; these operators have the appropriate substitutivity properties.

2.2. Modelling Transaction Systems. The automaton model is used to describe transaction systems; in fact, an interesting aspect of the modelling is that automata are used to describe *all* significant system components.

In particular, an automaton is used to model each *transaction* (and each *subtransaction*). An automaton is also used to model the *environment* of the system, which is the (possibly human) entity that generates the initial requests to perform top-level transactions. Other automata are used to described the *data objects*, often encapsulated with object-based concurrency control and recovery protocols. Finally, automata are used to describe other system components such as the *scheduler* component of a concurrency control protocol, which enforces global constraints on when various operations can be submitted to objects, and when various transactions can be created (i.e., started), committed or aborted. The entire transaction system is then described formally as the parallel composition of the component automata.

More specifically, a *transaction automaton* is an automaton T having an input action $CREATE(T)$, which awakens it, output actions $REQUEST\text{-}CREATE(T')$, each of which requests the creation of a child transaction T', input actions $REPORT\text{-}COMMIT(T', v)$ and $REPORT\text{-}ABORT(T')$, which report the fate of the child, commit or abort, to the parent transaction (in case of a commit, a return value is also provided), and output actions $REQUEST\text{-}COMMIT(T, v)$, by which the transaction announces that it has completed its work. See Figure 1 for the interface description.

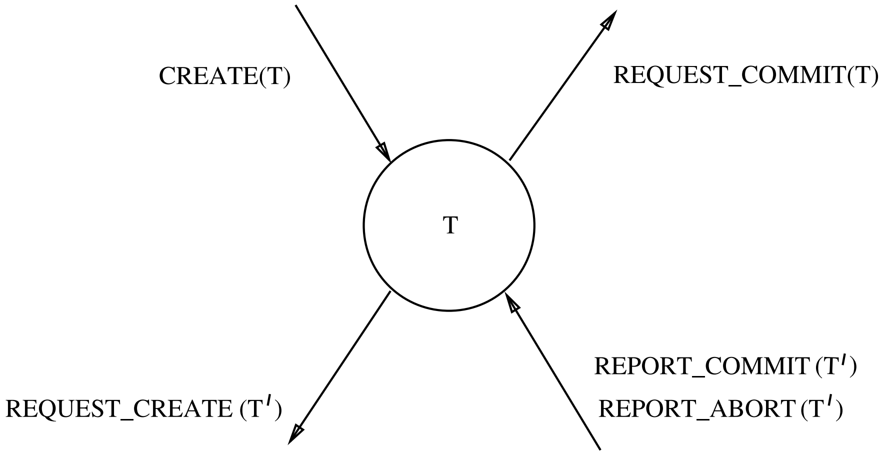

FIGURE 1. Interface for Transaction Automaton

The transaction has no significant further constraints – it is free, for example, to issue any number of requests to create subtransactions without waiting until previously-created children have returned. Or it can use information about the fate of previous children to decide what children to create next.

An *environment automaton* is similar to a transaction automaton in that it also requests to create "children" (in this case, the top-level transactions) and receives reports of their fates. In fact, we treat the environment formally as a special "root" transaction T_0.

A *data object automaton* receives inputs that are invocations of operations on the data object, and produces outputs that are the responses to those invocations. For uniformity with higher levels in the nested transaction hierarchy, the invocations are considered to be the lowest level $CREATE$ actions, analogous to the creation of a subtransaction, and the responses are considered to be the lowest level $REQUEST$-$COMMIT$ actions, analogous to the announcement by a subtransaction that it has completed. A data object automaton can also receive other inputs, to provide it with information that it can use to execute concurrency control and recovery protocols. See Figure 2 for a sample interface description. A *scheduler automaton* receives all the requests from all the other components as inputs, and issues decisions about when $CREATE$'s, $COMMIT$'s, $ABORT$'s and $REPORT$'s should occur.

3. Correctness

Now I describe the correctness condition to be satisfied by transaction systems. Informally, the requirement is that the system should look to its environment "as

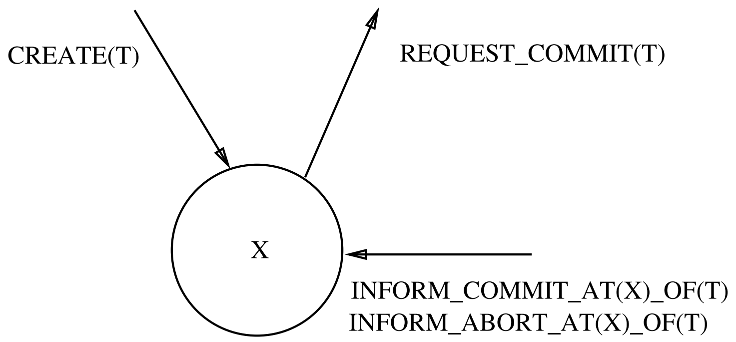

FIGURE 2. Interface for Object Automaton

if each set of sibling transactions executes strictly serially"; moreover, it should look "as if aborted transactions never performed any activity at all". In order to say this formally, it is nicest just to give a formal definition of a *serial system* – a transaction system in which the transactions do in fact execution serially, and in which aborted transactions never do perform any activity.

3.1. Serial Systems. A serial system is a transaction-processing system, as described above, in which the scheduler component is a *serial scheduler*, which runs siblings serially. The easiest way to describe this serial scheduler automaton is by giving explicit code in a simple guarded command language. In this representation, there is a piece of code for each action; in each case, the precondition is given as a predicate on the pre-state s, while the modifications are given by assignment statements. In this case, the state consists of the six indicated sets, all initially empty (except that *create-req* initially contains $\{T_0\}$); *completed* is an abbreviation for *committed* $\cup$ *aborted*. The code appears in Figure 3.

The interesting actions are $CREATE$ and $ABORT$. For example, an $CREATE$ action can only occur if a corresponding request has occurred and the $CREATE$ has not previously occurred; moreover, it cannot occur if the transaction was previously aborted. An $ABORT$ action can only occur under the same circumstances. These two actions together imply that no transaction is both created and aborted. Also, the serial scheduler does not create a transaction until each of the transaction's previously created sibling transactions has completed (i.e., committed or aborted).

3.2. Correctness Condition. Now it is easy to formulate the notion of correctness for any transaction-processing system A. Namely, it should be the case that any execution of A, there is an execution of the serial system that looks exactly the same to the system's environment. Here, "looks the same" is interpreted in terms of projection on the states and actions of the environment.

Stronger notions of correctness can also be considered, in terms of preserving the view of other system components besides the environment. For exam-

REQUEST_CREATE(T)
Effect:
$create\text{-}req := create\text{-}req \cup \{T\}$

REQUEST_COMMIT(T,v)
Effect:
$commit\text{-}req := commit\text{-}req \cup \{(T, v)\}$

CREATE(T)
Precondition:
$T \in create\text{-}req$
$T \notin created \cup aborted$
$siblings(T) \cap created \subseteq completed$
Effect:
$created := created \cup \{T\}$

ABORT(T)
Precondition:
$T \in create\text{-}req$
$T \notin created \cup aborted$
$siblings(T) \cap created \subseteq completed$
Effect:
$aborted := aborted \cup \{T\}$

COMMIT(T)
Precondition:
$(T, v) \in commit\text{-}req$
$T \notin completed$
Effect:
$committed := committed \cup \{T\}$

REPORT_COMMIT(T,v)
Precondition:
$T \in committed$
$(T, v) \in commit\text{-}req$
$T \notin reported$
Effect:
$reported := reported \cup \{T\}$

REPORT_ABORT(T)
Precondition:
$T \in aborted$
$T \notin reported$
Effect:
$reported := reported \cup \{T\}$

FIGURE 3. Transition Relation for the Serial Scheduler

ple, one might want to preserve the view of all transaction automata, or of all "non-orphan" transaction automata (i.e., those that do not have any aborted ancestors.

4. Algorithms

A very large amount of work has been carried out in the database and distributed systems design communities, in developing implementations of transactions. The most important algorithms are summarized briefly below. All of these algorithms are described in much greater detail in [**15**].

4.1. Locking. By far the most commonly-used algorithms are those based on some form of *locking*. In a locking algorithm, "locks" for data objects are kept with the objects themselves. A transaction that wishes to access the object must first secure an appropriate lock on the object. Depending on the data type of the object and on the particular protocol being used, there are different conventions about which transactions can hold locks simultaneously on the same object. Locking protocols are generally designed to make it appear to the environment as if the transactions ran in the order defined by the temporal order of their commit events.

A typical protocol for locking in a nested transaction system is described by the code in Figure 4, representing the automaton associated with data object X. This protocol allows X to be of an arbitrary data type; all that is assumed about the data type is that the correct *sequential executions* of the object (i.e., those in which invocations and responses strictly alternate) are known and are captured by the set $S(X)$; furthermore, there is a notion of *commutativity* of operations, which can be used to express the fact that two operations can be interchanged without upsetting correctness. The object is assumed to have two additional inputs, *INFORM-COMMIT* and *INFORM-ABORT*, by which it discovers that particular transactions (at any level) have committed or aborted, respectively.

Roughly speaking, the algorithm works as follows. The object X keeps track of a list of operations, $intentions(T)$, for every (potential) transaction T. This list represents the activity at X that has been performed by descendants of T that are known by object X to have committed "up to the level of" T. (That is, the descendants and all their ancestors up to T have committed.) The sequence $total(T)$ is then a shorthand for the concatenated sequence of intentions lists starting from the top level and down in a chain to T; thus, this list represents all the activity that should be known by T.

When the object gets informed that a transaction T has committed, it passes T's intentions list up to its parent (concatenating it at the end of what was there previously). When the object hears that a transaction has aborted, it simply discards its intentions. The object is allowed to perform an operation requested by T, provided that, roughly speaking, no non-ancestor U of T holds an "incompatible" lock, i.e., one that fails to commute with the new operation.

CREATE(T)
Effect:
$created := created \cup \{T\}$
INFORM_COMMIT_AT(X)**OF**(T)
Effect:
$intentions(parent(T)) := intentions(parent(T))intentions(T)$
$intentions(T) := \lambda$
INFORM_ABORT_AT(X)**OF**(T)
Effect:
$intentions(U) := \lambda, U \in desc(T)$
REQUEST_COMMIT(T,v)
Precondition:
$T \in created - commit\text{-}req$
there do not exist U, (T', v') such that
(T, v) does not commute with (T', v')
$(T', v') \in intentions(U)$
$U \notin ancestors(T)$
$perform(total(T)(T, v))$ is a behavior of S(X)
Effect:
$commit\text{-}req := commit\text{-}req \cup \{T\}$
$intentions(T) := (T, v)$

FIGURE 4. Transition Relation for General Commutativity-based Locking Object

In the code for *REQUEST-COMMIT* above, we interpret the condition that (T', v') is an element of $intentions(U)$ as meaning that U holds a "(T', v')-lock".

The use of the term "forward commutativity" rather than simply "commutativity" in the code indicates that there are some subtleties in the notion of commutativity. The subtleties have to do with the fact that some invocations can have more than one possible response, or possibly no response, defined. These subtleties are sorted out in [**15**].

The scheduler used in this locking algorithm is very nondeterministic, creating, committing, aborting and reporting the fates of transactions in any order, subject to basic well-formedness constraints. The complete locking system is modelled by the composition of the transaction, environment, object and scheduler automata. Correctness for this system says that its executions look like executions of the serial system to the environment, as well as to all non-orphan transactions.

Figure 5 contains a variant of the general locking protocol "optimized" for the case of read-write objects, This protocol is essentially due to Moss [**20**]. Note that only the objects differ from the corresponding components in the general locking protocol – the transaction, environment and scheduler automata are the same as in the general protocol.

4.2. Pseudotime Algorithms. Pseudotime-based protocols use a different strategy for establishing an apparent serial order for transactions. Namely, the scheduler assigns a *pseudotime interval*, an interval of real time, to each transaction before creating it. These intervals are assigned in a nested way, so that

CREATE(T)
  Effect:
    $created := created \cup \{T\}$
INFORM_COMMIT_AT(X)**OF**(T)
  Effect:
    if $T \in write\text{-}locks$ then
      $write\text{-}locks := write\text{-}locks - \{T\} \cup \{parent(T)\}$
      $val(parent(T)) := val(T)$
    if $T \in read\text{-}locks$ then
      $read\text{-}locks := read\text{-}locks - \{T\} \cup \{parent(T)\}$
INFORM_ABORT_AT(X)**OF**(T)
  Effect:
    $write\text{-}locks := write\text{-}locks - desc(T)$
    $read\text{-}locks := read\text{-}locks - desc(T)$
REQUEST_COMMIT(T,v), T a *read*
  Precondition:
    $T \in created - commit\text{-}req$
    $write\text{-}locks \subseteq ancestors(T)$
    $v = val(least(write\text{-}locks))$
  Effect:
    $commit\text{-}req := commit\text{-}req \cup \{T\}$
    $read\text{-}locks := read\text{-}locks \cup \{T\}$
REQUEST_COMMIT(T,v), T a *write*
  Precondition:
    $T \in created - commit\text{-}req$
    $write\text{-}locks \cup read\text{-}locks \subseteq ancestors(T)$
    $v =$ "OK"
  Effect:
    $commit\text{-}req := commit\text{-}req \cup \{T\}$
    $write\text{-}locks := write\text{-}locks \cup \{T\}$
    $val(T) := data(T)$

FIGURE 5. Transition Relation for Read/Write Locking Object

the intervals for children are disjoint from each other and are included in the interval of the parent. Objects use the pseudotimes of individual operations to sort out the proper order for performing them. This protocol is essentially due to Reed [**21**].

The pseudotime scheduler is modelled as a specific automaton that generates pseudotime intervals and associates them with transactions, and the system is modelled by the composition of the transaction, environment, object and scheduler automata. Again, correctness for system says that its executions look like executions of the serial system to the environment, as well as to all non-orphan transactions. The apparent serial order is just that of the pseudotimes.

4.3. Hybrid Algorithms. Hybrid algorithms combine ideas of locking and pseudotime-based algorithms to achieve more efficient concurrency control. As do locking algorithms, hybrid algorithms serialize transactions so that they appear to run in the order of their commit events. The difference is that the objects obtain additional information about the commit events that they do not receive in locking algorithms – not only the fact that the commits has occurred, but "timestamps" indicating the precise order in which the commits happened. This extra knowledge allows the objects to make better inferences about the commit order; this reduced uncertainty in turn allows the object to respond sooner in some cases in a hybrid algorithm than it could in a corresponding locking algorithm. This makes hybrid algorithms somewhat better candidates for efficient extension to the multiprocessor setting.

Code for the object X in a typical hybrid protocol appears in Figure 6; note the similarity to the code for the general locking objects. This time, a set *intset* of operations rather than a sequence *intentions* is associated with each transaction; again, these represent all the activity of the transaction's descendants that has committed to the level of the transaction. From each *intset*, we can derive a corresponding intentions list by ordering the set according to the known timestamps. The sequence *total* is then defined from the intentions lists as before. Note that the relationship between the two operations in the code for *REQUEST-COMMIT* is slightly different from before. It is now described in terms of a relation C, which is a symmetric *serial dependency relation*. This is a formal way of describing dependencies among operations, and is closely related to the definitions of commutativity; again, the exact relationship is discussed in [**15**].

The hybrid scheduler is a specific automaton that generates timestamps for transactions, and the system is again modelled as the composition. Correctness again says that the executions look like executions of the serial system to the environment, as well as to all non-orphan transactions.

Because of the extra timestamp information, the constraints used for the actions in the hybrid object are slightly weaker than in the locking object, which sometimes permits faster responses. For example, in the case of a *queue* type data

CREATE(T)
Effect:
$created := created \cup \{T\}$
INFORM_COMMIT_AT(X)**OF**(T,t)
Effect:
$intset(parent(T)) := intset(parent(T)) \cup intset(T)$
$intset(T) := \emptyset$
$time(T) := t$
INFORM_ABORT_AT(X)**OF**(T)
Effect:
$intset(U) := \emptyset$ for all $U \in desc(T)$
REQUEST_COMMIT(T,v)
Precondition:
$T \in created - commit\text{-}req$
there do not exist U, (T', v') such that
$((T, v), (T', v')) \in C$
$(T', v') \in intset(U)$
$U \notin ancestors(T)$
$perform(total(T)(T, v)) \in S(\mathrm{X})$
Effect:
$commit\text{-}req := commit\text{-}req \cup \{T\}$
$intset(T) := \{(T,v)\}$

FIGURE 6. Transition Relation for Hybrid Object

object, the commutativity-based locking protocol does not allow two *enqueue* operations to proceed concurrently, because it has no way of later resolving the eventual order in which they should be serialized. The hybrid algorithm allows both to proceed, and uses information in obtains later about the commits of the two transactions that invoked the *enqueue* operations to resolve the order.

4.4. Optimistic Algorithms. The book [**15**] also describes some "optimistic" concurrency control algorithms, omitted here. In general, in optimistic algorithms, the objects allow operations to proceed rather freely; any inconsistencies are resolved later, before transactions commit. Correctness for optimistic systems is typically weaker than that of non-optimistic systems; it says only that executions look like executions of the serial system to the environment.

4.5. Orphan Management Algorithms. In distributed systems, various factors, including node crashes and network delays, can result in *orphan transactions* – descendants of aborted transactions – continuing to run even though their results can no longer be used. Since locking, timestamp and hybrid systems all guarantee that the environment, as well as non-orphan transactions, cannot distinguish the system from a serial system, orphans do not cause any problem with the basic notion of correctness. However, orphans can be undesirable because they can waste resources and because they can see inconsistent states of the data (if they see results that depend on the abort of their ancestors); this can cause unanticipated behavior. The purpose of orphan management algorithms is to eliminate activity by orphans as soon as possible, and to prevent them from seeing inconsistent data states.

In [**15**], several orphan management algorithms are modelled and discussed, including those developed by Liskov et al. [**14**] and by Herlihy and McKendry [**10**]. The first is based on a "piggybacking" strategy that attaches relevant information about aborts to messages, thereby allowing a transaction to detect the abort of any of its ancestors before it can perform any operation that depends on this abort. The second is based on a virtual clock protocol. Again, both are modelled in the automaton framework.

4.6. Replicated Data Management Algorithms. In many cases, multiple copies of data objects are maintained for the purpose of obtaining higher availability; then there is the extra problem of keeping those copies consistent, so it looks to the environment as if there is only one copy of each object. An important strategy for achieving this aim is to use *quorum-based* methods, as originated by Gifford [**8**] and developed by many others. These methods have generally been designed for the special case of read-write objects. They involve designating sets of copies that form *read quorums* and *write quorums*, respectively, where any read quorum must intersect any write quorum. A read of an object proceeds by reading a read quorum, while a write requires writing to a write quorum. Quorums may change dynamically.

An interesting aspect of such protocols is that they can sometimes be described as *serial* systems: the original serial system A in which the leaves are accesses to the objects can be expanded into a new serial system B in which these logical accesses are replaced by *subtransactions* that perform the reads and writes by accessing the object copies. In reality, we are actually interested in concurrent versions of system B; such versions can be constructed, for example, using locking, timestamp, hybrid, or optimistic implementations of system B. Note that this description separates the issues of replication cleanly from the issues of concurrency control

5. Correctness Proofs

It is quite easy to model all of the algorithms described above (and many others) as composed systems of automata. Indeed, just having the precise correctness specifications and the precise algorithm descriptions is already a significant aid to understanding (and documenting) these algorithms. But the real payoff for all the modelling work is the ability to reason formally about the correctness of these algorithms. Complete proofs of all these algorithms are carried out in [**15**]; here, I just outline the methods that are used for carrying out the proofs.

The proofs are all carried out by hand, rather than with machine assistance. The algorithms are typically not finite-state machines, which rules out the possibility of using model-checking or other finite-state methods. Some of the proofs use refinement mappings and forward simulations; these should be automatable using current theorem-proving technology. Many of the other proofs, however, are based on specially tailored (ad hoc) arguments, which do not seem to fit any style for which mechanical aid will soon be of much help. (Of course, a higher-order theorem prover could always be used to verify the arguments step-by-step.)

Note that the fact that we are working with the semantic model rather than with specific languages and proof systems gives us the flexibility to use the wide variety of proof methods that seems to be needed.

Recall that this work is an extension of the classical theory of database concurrency control [**2**]. The main result of that theory is the *Serializability Theorem*, a simple theorem that asserts that a collection of transactions is implemented correctly provided that there is *no cycle* in the dependency relation that specifies which transactions must follow which others. This dependency relation is based on the execution of conflicting operations on common data objects.

Therefore, the first thing that we did was to prove a generalized version of the Serializability Theorem; we call ours the Atomicity Theorem. Its statement is not as simple as that of the classical theorem, because our setting is so much more general, including the complications of nesting, aborts, and arbitrary data types. However, in some special cases that substantially generalize those covered by the classical theory (including nesting and arbitrary data types, but assuming certain restrictions on the data access), the Atomicity Theorem implies a condition very close to the absence-of-cycles condition in the classical theory.

The statement of the Atomicity Theorem is roughly as follows. Consider a particular execution α of the transaction-processing system. Suppose that there exists a relation R that totally orders each set of sibling transactions, in a way that is consistent with the obvious event dependencies, and such that the *view compatibility condition* is satisfied: the "view" of execution α held by each object X, when reordered according to R, is an execution of the serial specification $S(X)$. Then α looks like a serial execution to the environment.

In order to see that this statement generalizes the classical Serializability Theorem, note that that theorem can be restated to assert the existence of a total ordering of transactions that is consistent with the orders of operations at each object.

The proof of this theorem is based on standard compositionality results for I/O automata. We show that, given the view compatibility condition, the execution projects to give a trace of *each component* of the serial system. The argument for the objects rests directly on the view compatibility condition, and the arguments for the transactions and environment are straightforward. The argument for the scheduler is, however, a complicated induction.

The correctness proofs for most of the algorithms (including locking, pseudotime, hybrid and optimistic algorithms) rest on the Atomicity Theorem. It would be possible to verify each algorithm by checking the view compatibility condition directly for each object; however, our proofs have more modularity than this, because we would like to preserve the possibility of "mixing and matching" different concurrency control algorithms for different objects in the same system. Thus, for each general type of algorithm (e.g., locking, pseudotime, hybrid), we define a condition at the object boundaries that implies the view compatibility condition, and that can be satisfied by a variety of implementations of the same general type. For locking, pseudotime and hybrid systems, the conditions are called *dynamic atomicity*, *static atomicity* and *hybrid atomicity*, respectively. These conditions are essentially locally-checkable versions of the view compatibility condition, given the specific information provided to the objects in the given type of system. It is straightforward to show that (in the context of the appropriate scheduler) each of these conditions implies the needed view compatibility condition.

It remains, then, to show that each of the specific objects, e.g., the locking and hybrid objects presented above, satisfies the appropriate boundary condition. In each case, this argument is a fairly difficult, ad hoc induction on the number of operations performed, using the various notions of commutativity of operations. We do not see how to systematize or simplify these arguments. One strategy we have followed is to use them to verify only the most general, nondeterministic versions (not necessarily the most efficient versions) of the objects. The idea is that we should only carry out these arguments once for each general kind of object (locking, etc.) and then base the proofs for special case and optimized versions of the objects on the correctness results for the nondeterministic objects.

Once we have verified the nondeterministic versions of the objects, we have an easy time verifying the correctness of the many variants (e.g., special cases and optimized versions). For example, the read-write locking object above can be proved to be an implementation of the general commutativity-based locking object, in the special case where the object is a read-write object. This proof is done using a fairly standard *forward simulation* or *possibilities mapping* argument. Such an argument involves setting up a correspondence between the states of the high-level algorithm and those of the implementation, and proving, using induction, that the correspondence is preserved by steps of the algorithm. (See, e.g., [**18**].) This is the sort of argument that is sufficiently stylized to admit computer assistance, though we have not done this work of mechanization.

Other proof methods are used for some of the other algorithms. For example, the orphan management algorithms are verified using partial ordering methods. Given an execution of a basic system without orphan management, a *dependency partial ordering* is defined for the events, saying which ones might depend on the prior occurrence of which others. The orphan management algorithm then prunes out certain actions that would depend, in the sense of this dependency ordering, on the abort of an ancestor. This pruning can be done explicitly, using piggybacking, or implicitly, using, e.g., logical clocks. Assuming that such bad dependencies are eliminated, the remaining execution looks to each transaction, orphan or not, as if it were a non-orphan in the basic system without orphan management. Hence, it cannot see inconsistent states of the data.

The proofs for the replicated data algorithms are quite interesting. As described above, the presentation of the algorithms has a neat decomposition, separating the issues of replica management and concurrency control. The replica management is expressed by subtransactions in a serial system B, replacing logical accesses in another serial system A. The fact that these two serial systems correspond in the right way is proved using fairly standard assertional reasoning, including forward simulation arguments. Once we have correctness of the serial system B, correctness of concurrent versions of B follows by the correctness, already proved, of the locking, pseudotime, hybrid, or optimistic algorithm used to manage the concurrency. Thus, the keys to the proof here are the nice problem decomposition, and the use of correctness results already proved for other concurrency control algorithms.

6. Conclusions

In this paper, I have outlined how we have formalized the notion of atomic transaction, including considerations of nesting, aborts, and general data types. We have defined the correctness conditions that transactions have to satisfy, have presented a wide variety of important concurrency control protocols, and have carried out complete correctness proofs. There are several other protocols that have been proved correct using this framework but that do not appear in the book [**15**]. These deal, in particular, with multigranularity locking [**9, 13**] and with recovery protocols [**3**].

My hope is that much of this work will carry over to the setting of multiprocessors. Transactions seem to be useful programming constructs for expressing the data-manipulation requirements of some typical large data-processing parallel applications such as the human genome project. Newer multiprocessor architectures look a great deal like distributed systems, with all the anomalies of timing and failures. Thus, it seems as though transactions should be useful in this setting.

Of course, there is much work to be done in determining *how* transactions are to be implemented in multiprocessors. Each individual transaction and subtransaction can run on a single processor. There will be processor allocation issues, since there will be a great deal of freedom in deciding when (and where) the various transactions in the nested transaction tree get created. The function of the scheduler module will probably be decomposed into pieces that are associated with different transactions (as it typically is in distributed systems).

The separate transaction managers can run on separate processors; these should be located near the processors on which the associated transactions are running. The concurrency control for each object might normally run on a single processor; however, when the object is itself a complicated data structure, clever special-case object implementations will probably be designed, running on a collection of processors. These implementations might be proved correct by showing that they satisfy the dynamic atomicity condition, or any of the other object boundary conditions.

Many new implementation considerations may arise for transactions in the multiprocessor setting. In this case, I hope that we have at least provided a formal framework that will make it easy for implementors to reason carefully about their implementation ideas, and even to prove formally that their algorithms are correct.

As I have already emphasized, our strategy of working in terms of a semantic model instead of a specific language gave us the maximum flexibility in carrying out our correctness proofs. This strategy should also give the results maximum applicability in new settings, involving new programming languages and architectures.

This work has been a very large case study in coordinated verification of concurrent algorithms – one of the largest that has so far been done. Such a coordinated study is valuable both for what it contributes to its application area, and for what it teaches about the formal modelling and proof techniques. I have tried in this paper to indicate many of the insights that have been gained both about transaction processing and about formal modelling and verification.

I believe that many other similar coordinated verification studies should be carried out. There are several subareas of the general area of parallel computation that have collections of related algorithms that could be studied in this way. For example, there is a general issue of obtaining strong coherence from memories with weak correctness conditions, using assumptions about access patterns by the software. Some initial work in verifying such protocols has been done by Gibbons et al [**7**], but there are many other cases that could be studied. For another example, there is a collection of work on implementation of strongly coherent shared memory using distributed networks, where the network is equipped with various communication capabilities such as atomic broadcast, atomic multicast, process groups, group communication and the like. A typical representative of this work is that of Kaashoek [**11**]; some preliminary work on verifying Kaashoek's protocols appears in [**5**]. For each of these subareas of parallel computation, it should be possible to establish a suitable infrastructure of definitions and basic theorems, building upon an underlying automaton model, and then to use the theory to verify many algorithms in a coordinated fashion. Doing this should provide great insights into the application areas.

Note that liveness has not been considered in this work; this is primarily because it is not clear that any interesting liveness claims can be made for distributed databases concurrency control algorithms. However, for other case studies, there may be interesting liveness claims to be proved. In such cases, note that it would be necessary to use a model that is more general than the simple automaton model used above (and in [**15**]). In some cases, the full I/O automaton model [**17**] provides sufficient machinery to express the needed liveness conditions. However, in some cases this will not be enough, so that a more general model such as that in [**6**] or the model underlying TLA [**12**] will be needed.

Note finally that considerations of real time have also been neglected in this work. There is a new body of work on *real-time databases* that has begun in the real-time systems research community. This work includes implementations of transactions that use real time in the concurrency control algorithms. It also includes extra considerations such as time deadlines for the completion of transactions. Modelling and verifying such algorithms will require a generalized semantic model that gives explicit representation to real time. Recent research has produced several candidate models, including [**19, 1, 12**]. These remain to be tested on such case studies.

References

1. Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Automata, Languages and Programming: Proceedings of the 17th ICALP*, Lecture Notes in Computer Science 443, pages 322–335. Springer-Verlag, 1990.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.
3. Ranjan Das. A formalization of distributed commit in data-processing systems. Bachelors Thesis, MIT Dept. of Electrical Engineering and Computer Science, June 1992. Also, [**4**].
4. Ranjan Das and Alan Fekete. Modular reasoning about open systems: A case study of distributed commit. In *Proceedings of Seventh International Workshop on Software Specification and Design*, pages 30–39, Los Angeles, CA, December 1993. Also, [**3**].
5. Alan Fekete, Frans Kaashoek, and Nancy Lynch. Implementing shared objects using multicast communication, February 1994. Submitted for publication.
6. Rainer Gawlick, Roberto Segala, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. In *Proceedings of the 21st ICALP*, July 1994. To appear. Also, Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 1993.
7. Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991. Also, AT&T Bell Laboratories, 11211-910509-09TM and 11261-910509-6TM, May, 1991.
8. D. Gifford. Weighted voting for replicated data. In *Proceedings of 7th ACM Symposium on Operating System Principles*, pages 150–162, December 1979.
9. J. Gray, R. Lorie, A. Putzulo, and J. Traiger. Granularity of locks and degrees of consistency in a shared database. Technical Report RJ1654, IBM, San Jose, CA, September 1975.
10. M. P. Herlihy and M. McKendry. Timestamp-based orphan elimination. *IEEE Transactions on Software Engineering*, 15(7):825–831, July 1989.
11. Marinus Frans Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Centrale Huisdrukkerij VRIJE Universiteit, Amsterdam, 1992.

12. Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, December 25 1991.
13. J. Lee and A. Fekete. Multi-granularity locking for nested transactions. In *Proceedings of 3rd Biannual Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *Lecture Notes in Computer Science*, pages 160–172. Springer-Verlag, New York, NY, May 1991.
14. B. Liskov, R. Scheifler, E. F. Walker, and W. E. Weihl. Orphan detection. In *Proceedings of 17th International Symposium on Fault-Tolerant Computing*, pages 2–7. IEEE, July 1987.
15. N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, 1994.
16. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, April 1987.
17. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Also, in Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November 1988.
18. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – Part I: Untimed systems. Submitted for publication. Also, MIT/LCS/TM-486.
19. Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446, Mook, The Netherlands, June 1991. Springer-Verlag. Also, MIT/LCS/TM-458.
20. J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1981. Also, Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, April 1981. Also, published by MIT Press, Cambridge, MA, March 1985.
21. D. P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Massachusetts Institute of Technology, 1978. Also, Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, October 1978.

Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

E-mail address: lynch@theory.lcs.mit.edu