Reasoning About Atomic Objects

Maurice P. Herlihy and Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890
U.S.A.[1]

Abstract

Atomic transactions are a widely-accepted technique for organizing activities in reliable distributed systems. In most languages and systems based on transactions, atomicity is implemented through atomic objects, which are typed data objects that provide their own synchronization and recovery. This paper describes new linguistic mechanisms for constructing atomic objects from non-atomic components, and it formulates proof techniques that allow programmers to verify the correctness of such implementations.

1. Introduction

A *distributed system* consists of multiple computers (called nodes) that communicate through a network. Programs written for distributed systems, such as airline reservations, electronic banking, or process control, must be designed to cope with failures and concurrency. Concurrency arises because each process executes simultaneously with those at other nodes as well as those at the same node, while failures arise because distributed systems consist of many independently-failing components. Typical failures include node crashes, network partitions, and lost messages.

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable and recoverable. *Serializability* [26] means that transactions appear to execute sequentially, and *recoverability* means that a transaction either succeeds completely or has no effect. A transaction's effects become permanent when it *commits*, its effects are discarded if it *aborts*, and a transaction that has neither committed or aborted is *active*.

[1] This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

In most languages and systems based on transactions, atomicity is implemented through *atomic objects*, which are typed data objects that provide their own synchronization and recovery. Languages such as Argus [19], Avalon [12], and Aeolus [32] provide a collection of primitive atomic data types, together with constructs for programmers to define their own atomic types. The most straightforward way to define a new atomic type is to use an existing atomic data type as a representation, but objects constructed in this way often support inadequate levels of concurrency [31]. Instead, it is often necessary to implement new atomic objects by carefully combining atomic and non-atomic components, and it is the responsibility of the programmer to ensure that the implementation is indeed atomic at the "right" level of abstraction.

In this paper,

- We describe new linguistic mechanisms for constructing atomic objects from non-atomic components. These mechanisms are currently being implemented as part of the Avalon [12] project at Carnegie Mellon.
- We formulate proof techniques that allow programmers to verify the correctness of atomic objects implemented using our mechanisms.

Although language and system constructs for implementing atomic objects have received considerable attention in the distributed systems community, the problem of verifying the correctness of programs that use those constructs has received surprisingly little attention. Techniques for reasoning about concurrent programs are well-known [1, 13, 18, 25], but are not adequate for reasoning about atomicity. They typically address issues such as mutual exclusion or the atomicity of individual operations; they do not address the more difficult problems of ensuring the serializability of arbitrary sequences of operations, nor do they address recoverability. Reasoning about atomicity is inherently more difficult than reasoning about concurrency alone.

We view the development of new linguistic mechanisms and proof techniques as complementary tasks. Verification techniques can serve not only as aids for reasoning about atomic objects, and hence about transaction-based distributed systems, but also as the foundation of a methodology for their design and implementation. This notion is analogous to Gries's contention that loop invariants and termination functions facilitate the development of programs-in-the-small [9], and Liskov and Guttag's similar contention that representation invariants and abstraction functions facilitate the development of programs-in-the-large [20].

This paper is organized as follows. In Section 2, we present our model and basic definitions, and in Section 3, we introduce and motivate the relevant language primitives provided by Avalon. In Section 4, we describe our verification techniques, which are illustrated by an extended example in Section 5. Section 6 concludes with a discussion and a brief overview of related work.

2. Model

The basic containers for data are called *objects*. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to create and manipulate objects of that type. For example, a file might provide Read and Write operations, and a FIFO queue might provide Enq and Deq operations.

A computation is modeled by a *history*, which is a finite sequence of *events*. An *invocation* event is written as *x op(args*) A*, where *x* is an object name, *op* an operation name, *args** a sequence of arguments, and *A* a transaction name. A *response* event is written as *x term(res*) A*, where *term* is a termination condition, and *res** a sequence of results. We use "Ok" for normal termination. A *commit* or *abort* event is written *x Commit A* or *x Abort A*, and it indicates that the object *x* has learned that transaction *A* has committed or aborted[2]. A response *matches* an earlier invocation if their object names agree and their transaction names agree. An invocation is *pending* if it has no matching response. An *operation* in a history is a pair consisting of matching invocation and response events. An operation p_0 *lies within* p_1 in H if the invocation event for p_1 precedes that of p_0 in H, and the response event for p_1 follows that of p_0.

A *transaction subhistory*, *H | A* (*H* at *A*), of a history *H* is the subsequence of events in *H* whose transaction names are *A*. *H | S* and *H | x* are defined similarly, where *S* is a set of transactions and *x* is an object. A history is *complete* if every invocation has a matching response. Let *complete(H)* denote the longest complete subhistory of *H*. Histories *H* and *G* are *equivalent* if *complete(H) | A = complete(G) | A* for all transactions *A*.

A history H is *well-formed* is it satisfies the following conditions for all transactions A:

1. The first event of H | A is an invocation.
2. Each invocation in H | A, except possibly the last, is immediately followed by a matching response or by an abort event.
3. Each response in H | A is immediately preceded by a matching invocation, or by an abort event.
4. If H | A includes a commit or abort event, it must be the last event.

A well-formed history *H* is *sequential* if:

1. Transactions are not interleaved. I.e., if any event of transaction *A* precedes any event of *B*, then all events of *A* precede all events of *B*.
2. All transactions, except possibly the last, have committed.

Each object has a *sequential specification* that defines a set of *legal* sequential histories for that object. This set is defined indirectly by using conventional specification techniques, e.g., the axiomatic style of Larch [10], that describe an object's values and pre- and postconditions on its operations. For example, the sequential specification for a FIFO queue object includes all and only histories in which items are enqueued and dequeued in FIFO order. A sequential history *H* involving multiple objects is *legal* if it is legal at each object, i.e., each subhistory *H | x* belongs to the sequential specification for *x*.

H is *atomic* if *H | committed(H)*, the subhistory associated with committed transactions, is equivalent to some legal sequential history. We focus here on "pessimistic" atomicity mechanisms, in which an active transaction with no pending invocations is always allowed to commit. *H* is *on-line atomic* if every well-

[2] Although Avalon permits transactions to be nested [24, 27], this paper considers only single-level transactions.

formed history *H'* constructed by appending *commit* events to *H* is atomic. Any sequential history equivalent to *H'* | *committed(H')* is called a *serialization* of *H*. *H* is on-line atomic if every one of its serializations is legal.

The only practical way to ensure atomicity in a decentralized distributed system is to have each object perform its own synchronization and recovery. Nevertheless, *H* is not necessarily atomic just because *H* | *x* is atomic for each object *x*. To ensure that all objects choose compatible serialization orders, it is necessary to impose certain additional restrictions on the behavior of atomic objects. Atomic objects in Avalon are subject to the restriction that transactions must appear to execute sequentially in the order they commit, a property that Weihl [30] has called *hybrid atomicity*. Hybrid atomicity is a local property, meaning that if each object is hybrid atomic, a system composed of hybrid atomic objects is itself hybrid atomic. Thus, under this restriction, it suffices to consider only object subhistories.

To capture this restriction, we make the following adjustments to our model. When a transaction commits, it is assigned a logical timestamp [17], which appears as an argument to that transaction's commit events. These timestamps determine the transactions' serialization order. Commit timestamps are subject to the following well-formedness constraint, which reflects the behavior of logical clocks: if *B* executes a response event after *A* commits, then *B* must receive a later commit timestamp. Henceforth, a history is atomic if its transactions are serializable in commit timestamp order, and it is on-line atomic if the result of appending commit events with well-formed commit timestamps is atomic.

q Enq(x) A
q Enq(y) B
q Ok() B
q Ok() A
q Commit(1:30) A
q Commit(1:15) B
q Deq() C
q Ok(y) C

q Enq(x) A
q Enq(y) B
q Ok() B
q Ok() A
q Commit(1:15) B
q Deq() C
q Ok(y) C

Figure 2-1: H_1 (left) is on-line atomic, but H_2 (right) is not.

For example, consider the two histories H_1 and H_2 for a FIFO queue *q* shown in Figure 2-1. H_1 is on-line atomic. It has two serializations: one in which *B* precedes *A*, and one in which *B* precedes *A* and *A* precedes *C*, and it is easily verified that both are legal. H_2, however, is atomic but not on-line atomic, since the history $H'_2 = H_2 \bullet$ *q Commit(1:00) A* • *q Commit(1:30) C* is not equivalent to any legal sequential queue history (*y* is dequeued out of order).

The use of commit-time serialization distinguishes Avalon from other transaction-based languages, which are typically based on some form of strict two-phase locking [5]. We chose to support commit-time serialization because it permits more concurrency than two-phase locking [30], as well as better availability for replicated data [11]. Because commit-time serialization is compatible with strict two-phase locking, applications that use locking can still be implemented in Avalon.

3. Language Constructs

Avalon is currently being implemented as extensions to C++ [29]. The basic language construct for implementing atomicity in Avalon is the *tid* (transaction identifier) data type. Tids are a partially ordered set of values. The two most important operations provided by tids are creation and comparison. The *creation* operation, written:

```
    tid t = *(new tid);
```

creates a new tid value, and the *comparison* operation, written:

```
    t1 < t2;
```

returns information about the order in which its arguments were created. If the comparison evaluates to *true*, then (1) every serialization that includes the creation of *t1* will also include the creation of *t2*, and (2) the creation of *t1* precedes the creation of *t2*. If *t1* and *t2* were created by distinct transactions *T1* and *T2*, then a successful comparison implies that *T1* is committed and serialized before *T2*, while if *t1* and *t2* were created by the same transaction, then *t1* was created first. If the comparison evaluates to *false*, then the tids may have the reverse ordering, or their ordering may be unknown. Comparison induces a partial order on tids that "strengthens" over time: if *t1* and *t2* are created by concurrent active transactions, they will remain incomparable until one or more of their creators commits. If a transaction aborts, its tids will not become comparable to any new tids.

Atomic objects in Avalon provide *Commit* and *Abort* operations that are called by the system as transactions commit or abort. Commit typically discards recovery information for the committing transaction, and Abort typically discards the tentative changes made by the aborting transaction. Both Commit and Abort have a tid argument, which is used as follows. If *t* is the argument to Commit, then any tid *t'* satisfying the predicate:

```
    isDesc(t',t)
```

was created by the committed transaction *t*.

The argument for Abort is defined analogously. Intuitively, Commit and Abort operations in Avalon are expected to affect liveness, but not safety. For example, delaying a Commit or Abort operation may delay other transactions (e.g., by failing to release locks) or reduce efficiency (e.g., by failing to discard unneeded recovery information), but it should never cause a transaction to observe an erroneous state[3].

An atomic object in Avalon is defined by a C++ class that inherits from the Avalon built-in class *atomic* (see [12]). Syntactically, a class is defined by a collection of *members*, which are the components of the object's representation, and a collection of operation implementations. With occasional minor variations, implementations of operations of atomic objects in Avalon have the following form:

```
    tid t = *(new tid);
    when(TEST)
      pinning()
        BODY;
```

[3] We do not address liveness properties in this paper, though certain ones are clearly of great interest. We rely on the extensive work on temporal logic, e.g., [23], for reasoning about liveness.

The **new** statement generates a new tid which is used to "tag" the current operation. The **when** statement is a conditional critical region: the statement enclosed by the **when** is executed when **TEST** evaluates to *true*. **TEST** is typically an expression comparing the operation's newly created tid to other tids embedded in the object's representation. When **TEST** evaluates to *true*, the operation can be executed without violating atomicity. To ensure proper crash recovery, as distinct from transaction recovery, an object may be modified only within statements of the form: **pinning() BODY**. The **pinning** statement is included here for completeness; we do not address crash recovery in this paper. **BODY** computes a result and updates the object's state.

4. Verification

This section outlines a verification methodology for implementations of atomic objects.

An *implementation* is a set of histories in which events of two objects, a *representation* (rep) object r of type Rep and an *abstract* object a of type Abs, are interleaved in a constrained way: for each history H in the implementation, (1) the subhistories H | r and H | a satisfy the usual well-formedness conditions; and (2) for each transaction A, each representation operation in H | A lies within an abstract operation. Informally, an abstract operation is implemented by the sequence of rep operations that occur within it.

Our correctness criterion for the implementation of an atomic object a is as follows: The object is atomic if for every history in its implementation, H | a is atomic. We typically do not require H | r to be atomic.

To show the correctness of an atomic object implementation, we must generalize techniques from the sequential domain. Let *Rep* be the implementation object's set of values, *Abs* the set of values of the (sequential) data type being implemented, and OP the sequential object's set of operations. The subset of *Rep* values that are legal values is characterized by a predicate called the *rep invariant*, *I*: *Rep* $\rightarrow$ *bool*. The meaning of a legal representation is given by an *abstraction function*, *A*: *Rep* $\rightarrow 2^{\mathrm{OP}^*}$, defined only for values that satisfy the invariant. Unlike abstraction functions for sequential objects [15] that map the rep value to a single abstract value, our abstraction functions map the rep value to a set of sequential histories of abstract operations.

Our basic verification technique is to show inductively that the following properties are invariant. Let *r* be an object state after accepting the history *H*, and let *Ser(H)* denote the set of serializations of *H*.

1. $\forall$ *S* in *A*(*r*), *S* is a legal sequential history, and
2. *Ser(H)* $\subseteq$ *A*(*r*).

These two properties ensure that every serialization of *H* is a legal sequential history, and hence that *H* is on-line atomic. Note that if we were to replace the second property with the stronger requirement that *Ser(H)* = *A*(*r*), then we could not verify certain correct implementations that keep track of equivalence classes of serializations. In the inductive step of our proof technique, we show the invariance of these two properties across a history's events, e.g., as encoded as statements in program text.

5. An Example: A Highly Concurrent FIFO Queue

In this section, we illustrate our verification technique by applying it to a highly concurrent atomic FIFO queue implementation. Our implementation is interesting for two reasons. First, it supports more concurrency than commutativity-based concurrency control schemes such as two-phase locking. For example, it permits concurrent Enq operations, even though Enq's do not commute. Second, it supports more concurrency than any locking-based protocol, because it takes advantage of state information. For example, it permits concurrent Enq and Deq operations while the queue is non-empty.

5.1. The Representation

Information about Enq and Deq invocations is recorded in the following data structures:

```
struct enq_rec {
  item* what;                              // item enqueued
  tid enqr;                                // who enqueued it
  enq_rec(tid t, item* x)                  // constructor
    {enqr = t; what = x;};
};
struct deq_rec {
  item* what;                              // item dequeued
  tid enqr;                                // who enqueued it
  tid deqr;                                // who dequeued it
  deq_rec(tid d, tid e, item* x)           // constructor
    {deqr = d;
     enqr = e;
     what = x;
    };
  };
```

The *enqr* component is a tid generated by the enqueuing transaction, *deqr* is a tid generated by the dequeuing transaction, *what* is a pointer to the enqueued item, and the last component defines a constructor operation for initializing the struct.

The queue is represented as follows:

```
class queue: public atomic {
  deq_stack deqd;                // Stack of dequeued items
  enq_heap enqd;                 // Heap of enqueued items
public:
  queue();                       // Create empty queue
  void enq(item*);               // Enqueue an item
  item* deq();                   // Dequeue an item
  void commit(tid);              // Called on commit
  void abort(tid);               // Called on abort
};
```

The *enqd* component is a *partially ordered queue* (or heap) of *enq_rec*'s, ordered by their *enqr* fields. The *deqd* component is a stack of *deq_rec*'s used to undo aborted Deq operations.

5.2. The Operations

If B is an active transaction, then we say A is *committed with respect to* B if A is committed, or if A and B are the same transaction. Enq and Deq must satisfy the following (informally stated) synchronization constraints to ensure atomicity. Transaction A may dequeue an item if (1) the most recent transaction to execute a Deq is committed with respect to A, and (2) there exists a unique oldest element in the queue whose enqueuing transaction is committed with respect to A. A may enqueue an item if the last item dequeued was enqueued by a transaction committed with respect to A.

Given these conditions, Enq and Deq are implemented as follows:

```
void queue::enq(item* x) {
  tid who = *(new tid);                           // Caller's tid
  when (deqd.empty() || deqd.top().enqr < who)
    pinning()                                     // Making update
      enqd.insert(enq_rec(who,x));                // Record enqueue
}

item* queue::deq() {
  tid who = *(new tid);                           // Caller's tid
  when ((deqd.empty() || deqd.top().deqr < who) &&
        enqd.top_exists() && enqd.top().enqr < who)
    pinning(){                                   // Making update
      enq_rec e = enqd.remove();                 // Move from enqueued heap...
      deqd.push(deq_rec(who,e.enqr,e.what)); // to dequeued stack.
      return e.what;
    }
}
```

Enq enters its critical region when the item most recently dequeued was enqueued by a transaction committed with respect to A. The enqueuer's tid and the new item are inserted in *enqd*. Deq enters its critical region when the most recent dequeuing transaction has committed with respect to the caller, and *enqd* has a unique oldest item. It removes the item from *enqd* and records it in *deqd*.

In addition to Enq and Deq operations, the queue provides Commit and Abort operations that are applied to the queue as transactions commit or abort:

```
void queue::commit(tid who) {
  when(true)                    // Always ok to commit.
    pinning()                   // Making update.
      // Discard any deq records.
      if (!deqd.empty() && deqd.top().deqr < who) deqd.reset();
}

void queue::abort(tid who) {
  when(true)                    // Always ok to abort.
    pinning(){                  // Making update.
      while (!deqd.empty()) {   // Undo aborted dequeues.
        deq_rec d = deqd.top();
        if (isDesc(d.deqr,who)) {// Dequeued by aborting transaction?
          enqd.insert(enq_rec(d.enqr,d.what)); // Put it back ...
          d = deqd.pop();}      // and discard deq record.
        else break;             // No more dequeues to undo.
      }
      enqd.discard(who);        // Undo aborted enqueues.
```

```
        }
    }
```

The commit operation discards deq records for committed transactions, and the abort operation discards enq and deq records for aborted operations.

5.3. Representation Invariant, Abstraction Function, and Proof Sketch

We start with a lemma about sequential queue histories. Let Q be a sequential queue history (not necessarily legal). Define the auxiliary functions $ENQ(Q)$ and $DEQ(Q)$ to yield the sequences of items enqueued and dequeued in Q:

$$\begin{aligned} &\mathrm{DEQ}(\Lambda) = \mathrm{emp} && \mathrm{ENQ}(\Lambda) = \mathrm{emp} \\ &\mathrm{DEQ}(Q \bullet \mathrm{Deq}(x)) = \mathrm{DEQ}(Q) \bullet x && \mathrm{ENQ}(Q \bullet \mathrm{Enq}(x)) = \mathrm{ENQ}(Q) \bullet x \\ &\mathrm{DEQ}(Q \bullet \mathrm{Enq}(x)) = \mathrm{DEQ}(Q) && \mathrm{ENQ}(Q \bullet \mathrm{Deq}(x)) = \mathrm{ENQ}(Q) \end{aligned}$$

Here, "Enq(x)" and "Deq(x)" are shorthand for Enq and Deq operations, "•" denotes concatenation, "emp" the empty sequence of items, and "Λ" the empty history.

When reasoning about serializations, we need a way to recognize when it is legal to insert an operation in the *middle* of a legal sequential history.

> Lemma 1: Let $Q = Q_1 \bullet Q_2$ be a legal sequential queue history, and let p be a queue operation. The sequential history $Q' = Q_1 \bullet p \bullet Q_2$ is legal if $DEQ(Q')$ is a prefix of $ENQ(Q')$.

This lemma indirectly characterizes the conditions under which queue operations may execute concurrently; an analogous lemma would be needed for any other data type to be verified.

5.3.1. Representation Invariant

The queue operations preserve the following representation invariant.[4] For all rep values r:

1. No item is present in both the *deqd* and *enqd* components:

   $(\forall\ d{:}\ \mathit{deq\_rec})\ (\forall\ e{:}\ \mathit{enq\_rec})\ (d \in r.\mathrm{deqd} \wedge e \in r.\mathrm{enqd} \Rightarrow e.\mathrm{what} \neq d.\mathrm{what})$

2. Items are ordered in *deqd* by their enqueuing and dequeuing tids:

   $(\forall\ d1, d2{:}\ \mathit{deq\_rec})\ \ d1 <_d d2 \Rightarrow (d1.\mathrm{enqr} < d2.\mathrm{enqr} \wedge d1.\mathrm{deqr} < d2.\mathrm{deqr})$

   where $<_d$ is the total ordering on *deq_rec*'s imposed by the *deqd* stack.

3. Any dequeued item must previously have been enqueued:

   $(\forall\ d{:}\ \mathit{deq\_rec})\ d \in r.\mathrm{deqd} \Rightarrow d.\mathrm{enqr} < d.\mathrm{deqr}.$

Our proof technique requires that we show the representation invariant is preserved across the implementation of each abstract operation. It is conjoined to the pre- and postconditions of each of the operations' specifications.

[4] For brevity, we assume items in the queue are distinct, an assumption that could easily be relaxed by tagging each item in the queue with a timestamp.

5.3.2. Abstraction Function

Let P be a set of tids. P is a *prefix set* if, for all tids t and t', if $t \in P \wedge t' < t$, then $t' \in P$.

> **Lemma 2:** If H is an on-line atomic history for a set of tids and S is a serialization of H, then the tids whose creation operations appear in S form a prefix set.

Define the auxiliary function *OPS(r, P)* to yield the partially ordered set of operations tagged by tids in P. The elements of *OPS(r, P)* are given by:

$$\begin{aligned}
&\{\mathrm{Enq(x)} \mid (\exists\ \mathrm{e}{:}\mathit{enq\_rec} \in \mathrm{r.enqd})\ \mathrm{e.what} = \mathrm{x} \wedge \mathrm{e.enqr} \in \mathrm{P} \vee \\
&\qquad\qquad (\exists\ \mathrm{d}{:}\mathit{deq\_rec} \in \mathrm{r.deqd})\ \mathrm{d.what} = \mathrm{x} \wedge \mathrm{d.enqr} \in \mathrm{P}\ \} \cup \\
&\{\mathrm{Deq(y)} \mid (\exists\ \mathrm{d}{:}\mathit{deq\_rec} \in \mathrm{r.deqd})\ \mathrm{d.what} = \mathrm{y} \wedge \mathrm{d.deqr} \in \mathrm{P}\}
\end{aligned}$$

Each operation is "tagged" with a tid (*e.enqr*, *d.enqr*, or *d.deqr*). These tids induce a partial order on the elements of *OPS(r, P)*.

Let S be a partially ordered set of operations, and S' a sequence of operations. S' is a ***linearization*** of S if *elements(S) = elements(S')* and *order(S)* $\subseteq$ *order(S')*.

$$A'(\mathrm{r, P}) = \{\mathrm{Q} \mid \mathrm{Q\ is\ a\ linearization\ of\ OPS(r, P)}\}$$

The abstraction function $A(r)$ is defined as the union of $A'(r, P)$ over all prefix sets P. Note that $A(r)$ typically includes more histories than *Ser(H)*.

5.3.3. Proof Sketch

The queue implementation is verified by showing inductively that every sequential history in *Ser(H)* lies in $A(r)$ and that every sequential history in $A(r)$ is legal. For brevity, we give an informal summary of our arguments here; the formal proofs for the Enq and Deq operations are in the next section.

Suppose the object completes an operation Enq(x) with tid t, carrying the accepted history H to H', and the representation r to r'. It is immediate from Lemma 2 that *Ser(H')* $\subseteq A(r')$. To show that every history in $A(r')$ is legal, let $Q' \in A(r')$. If Q' fails to satisfy the prefix property of Lemma 1, there must exist y in *DEQ(Q')* such that x precedes y in *ENQ(Q')*, implying that the Enq of x is serialized before the Enq of y. Let t' be the enqueuing tid for the item at the top of the *deqd* stack, and let t'' be the enqueuing tid for y. The **when** condition for Enq ensures that $t' < t$, and the rep invariant ensures that $t'' \leq t'$, hence that $t'' < t$, which is impossible if the Enq of x is serialized first.

Similarly, suppose the object completes an operation Deq(x) with tid t, carrying the representation r to r'. Let $Q = Q_1 \bullet Q_2 \in A(r)$ and $Q' = Q_1 \bullet \mathrm{Deq(x)} \bullet Q_2 \in A(r')$. The rep invariant and the first conjunct of the **when** condition for Deq ensure that x is not an element of *DEQ(Q)*, and the second conjunct then ensures that x is the first element of *ENQ(Q) – DEQ(Q)*. Together, they imply that *DEQ(Q') = DEQ(Q)* $\bullet$ x is a prefix of *ENQ(Q') = ENQ(Q)*, hence that Q is legal by Lemma 1.

If a commit or abort event carries the accepted history H to H', and the corresponding commit or abort operation carries r to r', we must show that (1) $A(r') \subseteq A(r)$, and (2) that no history in $A(r) - A(r')$ is in *Ser(H')*. Property 1 ensures that every sequential history in $A(r')$ is legal, and Property 2 ensures that no valid serializations are "thrown away." For Commit, we check that every discarded history is missing an operation of a committed transaction, and for Abort, we check that every discarded history includes an operation of an aborted transaction; either condition ensures that the discarded history is not an element of *Ser(H')*.

Naturally, this verification relies on properties of sequential queues. To verify an implementation of another data type, one would have to rely on a different set of properties, but the arguments would follow a similar pattern. The basic synchronization conditions are captured by a type-specific analog to Lemma 1, characterizing the conditions under which an operation can be inserted in the middle of a sequential history. The rep invariant and abstraction function define how the set of possible serializations is encoded in the representation, and an inductive argument is used to show that no operation, commit, or abort event can violate atomicity.

5.4. Proof of Correctness for Enqueue and Dequeue

We will show that if the prefix property (1) holds of all serializations $h \in A(\mathrm{r})$ at the invocation of the enqueue operation, it holds of all serializations $H' \in A(\mathrm{r}')$ at the point of return. In the following, for $H \in A(\mathrm{r})$, $H' \in A(\mathrm{r}')$, let $H = H_1 \bullet H_2$ and $H' = H_1 \bullet \mathrm{op} \bullet H_2$ such that $\forall\ \mathrm{p} \in H_1\ \neg(\mathrm{who} < \mathrm{tid(p)}) \wedge \forall\ \mathrm{p} \in H_2\ \neg(\mathrm{tid(p)} < \mathrm{who})$, where op is the enqueue or dequeue operation, as the case may be.

5.4.1. Enqueue

We decorate the enqueue operation with two assertions, one after the **when** condition, and one at the point of return.

```
void queue::enq(item* x) {
  tid who = *(new tid);                  // Caller's tid.
  when (deqd.empty() || deqd.top().enqr < who)

     WHEN: {∀y y ∈ elements(DEQ(h)) ⇒ tid(Enq(y)) < who}

    pinning()                            // Making update.
      enqd.insert(enq_rec(who,x));       // Record enq.

     POST: {DEQ(h') = DEQ(h)}
}
```

Proof: Case 1: The queue is empty. Trivial since the antecedent of WHEN is false.

Case 2: The queue is nonempty. Then let y be an item dequeued in H, which implies that the tid of the enqueue operation of y is ordered before who by the WHEN assertion. The enqueue operation must be in H_1 since (1) the tids of all enqueue operations of dequeued items are all ordered before that of deqd.top().enqr (by the rep invariant), which is ordered before who (by the **when** condition); and (2) who is not ordered before any operation in H_1 (by the definition of $H = H_1 \bullet H_2$). Since the enqueue operations of all dequeued items are in H_1,

$$\mathrm{DEQ}(H)\ \textit{prefix}\ \mathrm{ENQ}(H_1) \qquad (*)$$

At the point of return, let e = Enq(x)/Ok(). From POST we have that:

$\mathrm{DEQ}(H') = \mathrm{DEQ}(H)$, which by (*)
$\Rightarrow \mathrm{DEQ}(H')\ \textit{prefix}\ \mathrm{ENQ}(H_1)$
$\Rightarrow \mathrm{DEQ}(H')\ \textit{prefix}\ \mathrm{ENQ}(H_1 \bullet \mathrm{e} \bullet H_2)$
$\Rightarrow \mathrm{DEQ}(H')\ \textit{prefix}$ of $\mathrm{ENQ}(H')$.

5.4.2. Dequeue

Here is the annotated Deq operation:

```
item* queue::deq() {
  tid who = *(new tid);              // Caller's tid.
  when ((deqd.empty() || deqd.top().deqr < who) && // Check for conflict
       enqd.top_exists() && enqd.top().enqr < who)

    {WHEN: $\forall$ Deq operations $d$ in $h$ ($tid(d) < who \Rightarrow d$ in $H_1$)}

    pinning() {                      // Making update.
      enq_rec e = enqd.remove();// Transfer from enqueued heap...
      deqd.push(deq_rec(who,e.enqr,e.what)); // to dequeued stack.
      return e.what;
    }

    {POST: $DEQ(h') = DEQ(h) \bullet x \wedge ENQ(h') = ENQ(H_1) \bullet ENQ(H_2)$}
}
```

and the proof:

Proof: From the first conjunct of the **when** condition and the second clause of the rep invariant, we know that DEQ(H) = DEQ(H_1). The second conjunct implies that there exists some x = *first*(ENQ(H) – DEQ(H)), the first item in the sequence of enqueued items that have not yet been dequeued. The third conjunct implies that this item, x, is in H_1. Thus, by properties on sequences, there exists some x = *first*(ENQ(H_1) – DEQ(H_1)).

At the point of return, let d = Deq()/Ok(x). POST implies that

DEQ(H_1 • d) *prefix* ENQ(H_1 • d)
⇒ DEQ(H') *prefix* ENQ(H_1 • d)
⇒ DEQ(H') *prefix* ENQ(H_1 • d • H_2)
⇒ DEQ(H') *prefix* ENQ(H').

6. Discussion and Related Work

Atomicity has long been recognized as a basic correctness property within the database community [2]. More recently, several research projects have chosen atomicity as a useful foundation for general-purpose distributed systems, including Avalon [12], Argus [19], Aeolus [32], and TABS [28][5]. Of these projects, however, only Avalon and Argus provide linguistic support for programmers to design and implement user-defined atomic data types, which Weihl and Liskov [31] argue is necessary for building large, realistic systems. To our knowledge, Avalon is the only language project to address the issue of verifying implementations of atomic objects.

Part of the Avalon design philosophy is that verification is facilitated by making constructs for synchronization and recovery as explicit as possible. For example, the tid data type makes the set of serializations directly observable to programs, and our example verification relies heavily on the properties of this built-in data type. Similarly, explicit user-defined commit and abort operations provide direct control over transaction recovery. By contrast, Argus relies on the programmer to encode

[5] EXODUS [6] and Dixon and Shrivastava's language [4], like Avalon, extend C++ to support recoverability, but neither gives programmers control over serializability.

information about the set of serializations in "atomic variants," treating commit and abort processing as a side-effect of normal operations. For a more detailed evaluation of the implicit approach in Argus, see Greif et al. [8].

The axiomatic approach for program verification is particularly well-suited for "syntax-directed" verification of sequential and concurrent programs. Axiomatic proof techniques originated with Hoare's axioms [14] for sequential program statements and were later extended for abstract data types by introducing abstraction functions [15], representation invariants, and data type induction rules. The axiomatic approach was also extended to shared-memory models of concurrent programs [25], and to message-passing models of distributed programs [1, 16]. One of the principal conclusions of our work is that such "pure" syntax-directed axiomatic methods seem poorly suited for reasoning about atomicity. In the sequential and concurrent domains, an object's state can be given by a single value, and each new operation simply transforms one value to another as prescribed by the appropriate axiom. Auxiliary variables are used to keep track of history information and the states (e.g., program counters) of concurrent processes. In the transactional domain, however, an atomic object's state must be given by a *set* of possible serializations, and each new operation is inserted somewhere "in the middle" of certain serializations (see Lemma 1). This distinction between physical and logical ordering is easily expressed in terms of reordering histories, but seems awkward to express axiomatically, i.e., using assertions expressed in terms of program text alone. Of course, the proofs in this paper could be axiomatized simply by encoding the set of serializations as auxiliary data, but we have found the resulting proofs complex and unnatural.

Best and Randell [3], Weihl [30], and Lynch and Merritt [22] have proposed formal models for transactions and atomic objects. Best and Randell use *occurrence graphs* to define the notion of atomicity, to characterize interference freedom, and to model error recovery. Their model does not exploit the semantics of data, focusing instead on event dependencies. Weihl's model is similar to ours: atomic objects are defined in terms of state machines and computations are modeled as histories. Lynch and Merritt model nested transactions and atomic objects in terms of *I/O automata*, which have been used to prove correctness of general algorithms for synchronization and recovery [7, 21]. None of these models were intended for reasoning about individual programs.

Our technique lies between "pure" syntax-directed axiomatic approaches and model-oriented operational ones. Because we wish to reason about specific programs, not abstract algorithms, our approach relies on annotating program text with assertions. Our assertion language, however, refers to operations, histories, and sets of histories directly, making it richer and more expressive than the usual first-order logic-based assertion languages. We have found our approach more natural for reasoning about transaction-based distributed systems where serializability and recoverability cannot be treated as independent properties.

References

[1] K.R. Apt, N. Francez, and W.P. DeRoever.
A Proof System for Communicating Sequential Processes.
ACM Transactions on Programming Languages and Systems 2(3):359-385, July, 1980.

[2] P.A. Bernstein and N. Goodman.
A survey of techniques for synchronization and recovery in decentralized computer systems.
ACM Computing Surveys 13(2):185-222, June, 1981.

[3] E. Best and B. Randell.
A Formal Model of Atomicity in Asynchronous Systems.
Acta Informatica 16(1):93-124, 1981.

[4] G. Dixon and S.K. Shrivastava.
Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems.
In *Proceedings of the 6th Symposium in Reliability in Distributed Software and Database Systems*. March, 1987.

[5] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The Notion of Consistency and Predicate Locks in a Database System.
Communications ACM 19(11):624-633, November, 1976.

[6] J.E. Richardson and M.J. Carey.
Programming Constructs for Database System Implementation in EXODUS.
In *ACM SIGMOD 1987 Annual Conference*, pages 208-219. May, 1987.

[7] M.P. Herlihy, N.A. Lynch, M. Merritt, and W.E. Weihl.
On the correctness of orphan elimination algorithms.
In *17th Symposium on Fault-Tolerant Computer Systems*. July, 1987.
Abbreviated version of MIT/LCS/TM-329.

[8] I. Greif, R. Seliger, and W.E. Weihl.
Atomic Data Abstractions in a Distributed Collaborative Editing System.
In *Proceedings of the 13th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 160-172. January, 1986.

[9] D. Gries.
The Science of Programming.
Spinger-Verlag, New York, 1981.

[10] J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, September, 1985.

[11] M.P. Herlihy.
A quorum-consensus replication method for abstract data types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[12] M.P. Herlihy and J.M. Wing.
Avalon: Language Support for Reliable Distributed Systems.
In *The Seventeenth International Symposium on Fault-Tolerant Computing*, pages 89-94. July, 1987.
Also available as CMU-CS-TR-86-167.

[13] M.P. Herlihy and J.M. Wing.
Axioms for concurrent objects.
In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 13-26. January, 1987.

[14] C.A.R. Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12(10):576-583, October, 1969.

[15] C.A.R. Hoare.
Proof of Correctness of Data Representations.
Acta Informatica 1(1):271-281, 1972.

[16] J. Hooman and W.-P. de Roever.
The Quest Goes On: A Survey of Proof Systems For Partial Correctness of CSP.
Lecture Notes in Computer Science 224: Current Trends in Concurrency.
Springer-Verlag, 1986, pages 343-395.

[17] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[18] L. Lamport.
Specifying Concurrent Program Modules.
ACM Transactions on Programming Languages and Systems 5(2):190-222, April, 1983.

[19] B.H. Liskov, and R. Scheifler.
Guardians and actions: linguistic support for robust, distributed programs.
Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[20] B.H. Liskov and J.V. Guttag.
Abstraction and Specification in Program Development.
The MIT Press, 1986.

[21] N. Lynch.
A Concurrency Control For Resilient Nested Transactions.
Technical Report MIT/LCS/TR-285, Laboratory for Computer Science, 1985.

[22] N. Lynch and M. Merritt.
Introduction to the Theory of Nested Transactions.
In *Proceedings of the International Conference on Database Theory.* Rome, Italy, September, 1986.
Sponsored by EATCS and IEEE.

[23] Z. Manna and A. Pnueli.
Verification of concurrent Programs, Part I: The Temporal Framework.
Technical Report STAN-CS-81-836, Dept. of Computer Science, Stanford University, June, 1981.

[24] J.E.B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, April, 1981.

[25] S. Owicki and D. Gries.
Verifying Properties of Parallel Programs: An Axiomatic Approach.
Communications of the ACM 19(5):279-285, May, 1976.

[26] C.H. Papadimitriou.
The serializability of concurrent database updates.
Journal of the ACM 26(4):631-653, October, 1979.

[27] D.P. Reed.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[28] A.Z. Spector, J. Butcher, D.S. Daniels, D.J. Duchamp, J.L. Eppinger, C.E. Fineman, A. Heddaya, and P.M. Schwarz.
Support for Distributed Transactions in the TABS prototype.
IEEE Transactions on Software Engineering 11(6):520-530, June, 1985.

[29] B. Stroustrup.
The C++ Programming Language.
Addison Wesley, 1986.

[30] W.E. Weihl.
Specification and implementation of atomic data types.
Technical Report TR-314, Massachusetts Institute of Technology Laboratory for Computer Science, March, 1984.

[31] W.E. Weihl, and B.H. Liskov.
Implementation of resilient, atomic data types.
ACM Transactions on Programming Languages and Systems 7(2):244-270, April, 1985.

[32] C.T. Wilkes and R.J. LeBlanc.
Rationale for the design of Aeolus: a systems programming language for an action/object system.
Technical Report GIT-ICS-86/12, Georgia Inst. of Tech. School of Information and Computer Science, Dec, 1986.