Reprinted from:
INTERNATIONAL JOURNAL OF PARALLEL PROGRAMMING

Vol. 18, No. 5, October 1989
Printed in Belgium

Verifying Atomic Data Types

Jeannette M. Wing[1]

Received July 1989; Revised March 1990

Atomic transactions are a widely-accepted technique for organizing computation in fault-tolerant distributed systems. In most languages and systems based on transactions, atomicity is implemented through atomic objects, typed data objects that provide their own synchronization and recovery. Hence, atomicity is the key correctness condition required of a data type implementation. This paper presents a technique for verifying the correctness of implementations of atomic data types. The significant aspect of this technique is the extension of Hoare's abstraction function to map to a set of sequences of abstract operations, not just to a single abstract value. We give an example of a proof for an atomic queue implemented in the programming language Avalon/C++.

KEY WORDS: Atomicity; program verification; fault-tolerance; transactions; distributed systems; abstract data types.

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 other processes on the local node and processes on remote nodes, 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. Informally, *serializability*[1] means that concurrent trans-

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890.

0885-7458/89/1000-0315$06.00/0

actions 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 nor aborted is *active*.

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,[2] Avalon,[3] and Aeolus[4] provide a collection of primitive atomic data types, together with constructs allowing 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.[5] Instead, programmers could implement new atomic objects by carefully combining atomic and non-atomic components and exploiting the semantics of the data type to provide more concurrency. This degree of freedom comes with a price: the programmer is now responsible for proving that the implementation of the user-defined data type is indeed atomic.

In this paper, we formulate proof techniques that allow programmers to verify the correctness of atomic objects. 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. To our knowledge, the Avalon Project conducted at Carnegie Mellon University is the only language project to address this particular program verification problem.

Techniques for reasoning about concurrent programs are well-known,[6–9] 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.

Our work distinguishes us from most other formal specification and verification research in concurrent and distributed systems since the presence of failures is addressed as seriously as the presence of concurrency and distribution. Our particular approach also distinguishes our work from many others: we focus on behavior and correctness of objects in a system and not on the processes (transactions) that manipulate them. We base the proof of correctness of the entire system on a local property of the objects in the system; if the property holds for each object, the correctness of the entire system is guaranteed. Thus, the problem of proving an entire distributed system correct is transformed into the more manageable problem of proving each of the objects in the system correct.

This paper is organized as follows. Section 2 describes our model and basic definitions, and illustrates most of them through simple examples. Section 3 defines three pieces in our verification technique, the most important of which is an extension of Hoare's abstraction function for data implementations. Section 4 introduces and motivates relevant Avalon/C++ programming language primitives. Section 5 gives an extended example using these primitives and a correctness proof following the technique outlined in Section 3. Section 6 discusses related work, in particular contrasting the particular correctness condition used with another more conventional one and contrasting our extended abstraction function with other kinds of mappings. Finally, Section 7 summarizes relevant current and future work.

2. MODEL FOR TRANSACTION-BASED DISTRIBUTED SYSTEMS

A distributed system is composed of a set of transactions and a set of objects. A *transaction* corresponds to a sequential process. We disallow concurrency within a transaction, but allow for multiple transactions to execute concurrently. *Objects* contain the state of the system. Each object has a *type*, which defines a set of possible *values* and a set of *operations* that provide the only means to create and manipulate objects of that type. A transaction can either complete successfully, in which case it *commits*, or unsuccessfully, in which case it *aborts*. We use the term *termination* for the end of the execution of an operation and *completion* for the end of the execution of a transaction.

Typically, a transaction executes by invoking an operation on an object, receiving results when the operation terminates, then invoking another operation on a possibly different object, receiving results when it terminates, etc. It then commits or aborts.

Although Avalon permits transactions to be nested,[10,11] the model presented here and our subsequent discussion consider only single-level transactions. Nested transactions provide a means to obtain concurrency within a transaction; Lynch and Merritt[12] present a formal model of nested transactions based on I/O automata. Our model of transactions borrows heavily from Weihl's, first described in his 1984 Ph.D. thesis[13] and more recently in Ref. 14.

2.1. Histories

We model a computation as a *history*, which is a finite sequence of events. There are four kinds of events: invocations, responses, commits, and aborts. 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** is a sequence of results. We use "Ok" for normal termination. A *commit* or *abort* event is written x *Commit* A or x *Abort* A; it indicates that the object x has learned that transaction A has committed or aborted, and we say "A has committed (or aborted) at x."

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 op_0 *lies within* op_1 in the history H if the invocation event for op_1 precedes that of op_0 in H, and the response event for op_1 follows that of op_0. For sequences, we use "$\bullet$" to denote concatenation, and "Λ" the empty sequence.

For a history H, we define *committed*(H) to be the set of transactions in H that commit in H at some object, and *aborted*(H) to be the set of transactions that abort in H at some object. We define *completed*(H) to be *committed*(H) $\cup$ *aborted*(H), and *active*(H) to be the set of transactions in H not in *completed*(H). Note that we can model a failure event (e.g., node crash) with abort events. A history H is failure-free if *aborted*$(H) = \varnothing$.

Example

The following history, H_1, involves two queue objects p and q, and four transactions A, B, C, and D:

$$
\begin{array}{lll}
p & Enq(1) & A \\
p & Enq(2) & B \\
p & Ok(\ ) & B \\
q & Enq(4) & B \\
p & Ok(\ ) & A \\
q & Ok(\ ) & B \\
p & Commit & B \\
q & Commit & B \\
p & Enq(3) & A \\
p & Ok(\ ) & A \\
p & Abort & A \\
p & Deq(\ ) & C \\
p & Ok(2) & C \\
q & Enq(5) & D \\
p & Enq(6) & C \\
p & Ok(\ ) & C
\end{array}
$$

The first event in H_1 is the invocation of the *Enq* operation on object p by transaction A. The fifth event is the matching response event. The seventh and eighth events indicate that p and q respectively have learned that B has committed; the eleventh indicates that p has learned that A has aborted. The *Enq* operation of 2 by B lies within the *Enq* of 1 by A.

A and B execute concurrently and both eventually complete, A unsuccessfully and B successfully. H_1 is not failure-free. C and D execute concurrently and are both active (have neither committed nor aborted) at the end of H_1. Hence, $committed(H_1) = \{B\}$, $aborted(H_1) = \{A\}$, $completed(H_1) = \{A, B\}$, and $active(H_1) = \{C, D\}$. When C dequeues from p, it receives 2. D's invocation of Enq on q is pending since there is no matching response event. Note that a transaction completes at most once, but a history records the completion by multiple events, one for each object with which the transaction is involved, e.g., as in B's two commit events.

H_1 shows an example of an atomic (to be formally defined) or intuitively "correct" history. H_1 is correct because there is some ordering on nonaborted transactions that is "equivalent" to a "sequential" version of H_1 and because A's effects are ignored. It would have been incorrect for C to dequeue 1 from p since A aborts. If A were to commit instead, then it would be correct either to have C dequeue 1, by ordering A before B, or to have C dequeue 2, by ordering B before A. Notice that a transaction can perform more than one operation, possibly on different objects. A performs two *Enq*'s on p and B performs one each on p and q. The intuition we would like to capture in our formal definitions is as follows: at the end of H_1 (1) p's first and only element is either 2 (C aborts) or 6 (C commits); and (2) q's first and only element is 4 (q does not have 5 in it because D's invocation is pending, yet it definitely has 4 in it because B's commit precedes D's invocation).

End example

A *transaction subhistory*, $H|A$ (H at A), of a history H is the subsequence of events in H whose transaction names are A. $H \mid S$ and $H \mid x$ are defined similarly, where S is a set of transactions and x is an object.

The following well-formedness conditions capture the requirement that each transaction performs a sequence of operations. If it then commits, it does nothing else afterwards.

Definition 1. A history H is *well-formed* if it satisfies the following conditions for all transactions A:

1. The first event of $H \mid A$ is an invocation.

2. Each invocation in $H \mid A$, except possibly the last, is immediately followed by a matching response or by an abort event.
3. Each response in $H \mid A$ is immediately preceded by a matching invocation, or by an abort event.
4. If $H \mid A$ includes a commit event, no invocation or response event may follow it.
5. A transaction can either commit or abort, but not both, i.e., $committed(H) \cap aborted(H) = \varnothing$.

These constraints imply that a transaction cannot invoke one operation on an object x and then another on x (or any other object) without first receiving a response from its first invocation. If a transaction commits, it cannot have any pending invocations; if it aborts, it may be in the middle of executing an operation, and thus have at most one pending invocation. Once a transaction commits, it cannot perform further operations. As in Weihl's model, we permit a transaction to continue to invoke operations after it has aborted. We have two reasons for placing few restrictions on aborted transactions: first, subsequent relevant definitions will ensure that we never deal with any events that involve aborted transactions; second, additional restrictions might be too strong to model systems with orphans[32,16] and hence, make our model unnecessarily restrictive. Finally, a transaction is not allowed to commit at some objects and abort at others. Henceforth, we assume all histories are well-formed.

Definition 2. A history H is *sequential* if:

1. Transactions are not interleaved. That is, if some event of transaction A precedes some event of B, then all events of A precede all events of B.
2. All transactions, except possibly the last, have committed.

The second condition implies that the last transaction in H may possibly be active, and hence eventually commit or abort.

Examples

H_1 is well-formed. $H_1 \mid B$ is the transaction subhistory:

$$\begin{array}{lll} p & Enq(2) & B \\ p & Ok(\,) & B \\ q & Enq(4) & B \\ q & Ok(\,) & B \\ p & Commit & B \\ q & Commit & B \end{array}$$

and $H_1 \mid p$ is the object subhistory:

p	*Enq*(1)	A
p	*Enq*(2)	B
p	*Ok*( )	B
p	*Ok*( )	A
p	*Commit*	B
p	*Enq*(3)	A
p	*Ok*( )	A
p	*Abort*	A
p	*Deq*( )	C
p	*Ok*(2)	C
p	*Enq*(6)	C
p	*Ok*( )	C

The following well-formed subhistory of H_1 is sequential:

p	*Enq*(2)	B
p	*Ok*( )	B
q	*Enq*(4)	B
q	*Ok*( )	B
p	*Commit*	B
q	*Commit*	B
p	*Deq*( )	C
p	*Ok*(2)	C
p	*Enq*(6)	C
p	*Ok*( )	C

End examples

Informally, two histories H and G are equivalent if for each transaction A, ignoring pending invocations, A performs the same events in the same order in H as in G.

Definition 3. Let *terminated*(H) denote the longest subhistory of H such that every invocation has a matching response. Histories H and G are *equivalent* if $\mathit{terminated}(H) \mid A = \mathit{terminated}(G) \mid A$ for all transactions A.

If H and G are equivalent, then for all objects x the value of x after H should be the same as that of x after G; the converse is not true.

We will be primarily interested in histories that are "complete" in the sense that every transaction either completes at every object with which it participates or does nothing at the object.

Definition 4. A history H is *complete* if for all objects x and transactions A, either $A \in \mathit{completed}(H \mid x)$ or $H \mid A \mid x = A$.

H_1 is not complete but $H_1 \mid A$ and $H_1 \mid B$ are.

2.2. Specifications

We model a system's *behavior* as a set of histories. A system's behavior is determined by the behavior of its components, i.e., objects and transactions, and their interactions. A component's specification constrains the component's behavior and indirectly constrains the system's behavior. We assume that the behavior of each system component c is given by a set, denoted by $c.behavior$, of complete histories involving only that component. Given the behavioral specifications of a system's components, the possible behavior of the system is then all complete histories H such that, for all transactions A and objects x, $H \mid A \in A.behavior$ and $H \mid x \in x.behavior$. Note that a component's specification constrains the occurrence of events in which it participates and places no constraints on the occurrence of other events.

Our verification method is based on showing the correctness of the implementations of objects with respect to an object's behavior (set of histories). In order to define correctness (i.e., atomicity) in terms of sets of histories, we find it natural and convenient to define it through an object's *sequential specification*, which captures the object's sequential failure-free behavior. We denote the sequential specification of an object x by $x.sequential$. In this section, we first define state machines used to describe sequential specifications for objects and then the notion of a *legal* sequential history, which we use to define atomicity in the next section.

2.2.1. State Machines: Sequential Specifications for Objects

Each object has a *sequential specification*, which defines its behavior in the absence of concurrency and failures. An object's sequential specification is a set of operation sequences. We use state machines to describe an object's sequential specification. A *state machine* is a four-tuple $\langle \text{STATE}, s_0, \text{OP}, \delta \rangle$, where STATE is a state domain, $s_0 \in \text{STATE}$ is an initial state, OP is a set of operations ("state transitions") $\delta: \text{STATE} \times \text{OP} \rightarrow \text{STATE}$ is a partial *transition function*. We extend the domain of the transition function to finite sequences of operations in the usual way. If s is a state, p is an operation, ops is a finite sequence of operations, and $\perp$ denotes "undefined," then $\delta^*: \text{STATE} \times \text{OP}^* \rightarrow \text{STATE}$ is defined as follows:

$$\delta^*(s, \Lambda) = s$$
$$\delta^*(\perp, ops) = \perp$$
$$\delta^*(s, ops \bullet p) = \delta(\delta^*(s, ops), p)$$

An operation sequence ops is *defined in* state s if $\delta^*(s, ops) \neq \perp$. An operation sequence ops is *accepted* by a machine, $\langle \text{STATE}, s_0, \text{OP}, \delta \rangle$, if ops

is defined in s_0. The set of sequences accepted by a state machine M is the language of M.

To define an object's sequential specification by a state machine, we let the machine's set of operations be the object's set of operations, and we choose the state domain and δ such that the language of the machine is the desired set of operation sequences. We further interpret a machine state to contain a mapping from the object to a value, drawn from the object's domain of values. We use an operation in an operation sequence to represent a single execution of that operation. Operation executions change the values of objects. We denote an operation on object x as $x\ op(args^*)/term(res^*)$, thus pairing the invocation and response events in the obvious way. [We can ignore transaction names since we are in the sequential domain here.]

Examples

The following operation sequence would be in $q.sequential$:

$$q \quad Enq(1)/Ok(\,)$$
$$q \quad Deq(\,)/Ok(1)$$

Here, q's value changes from being the empty queue (at the beginning of the sequence) to the queue with only 1 in it, and then back to the empty queue (at the end of the sequence). The following operation sequence would not be in $q.sequential$:

$$q \quad Enq(1)/Ok(\,)$$
$$q \quad Deq(\,)/Ok(2)$$

End examples

In these examples, we appealed to the reader's intuition as to what the "correct" sequential behavior for queues would be. To be more concrete in this paper, we use notation from the Larch Family of Specification Languages[17] to describe state machines, and hence sequential specifications for objects. Other specification approaches (e.g., VDM,[18] Z,[19] I/O automata[20]) would be just as appropriate.

Larch interface specifications describe the object's value in the machine's initial state and the machine's state transitions. Figure 1 gives interface specifications for FIFO sequential queues. A queue is **initially** empty. The transition function is described by giving, for each operation (Enq and Deq), a precondition (**requires** clause) describing the set of states in which it is defined, and a postcondition (**ensures** clause) describing the relationship between the object's value in the invocation state and its value in the termination state. An unprimed argument formal, e.g., q, in a

```
initially q = emp

Enq(e)/Ok()
   requires true
   modifies at most (q)
   ensures q' = ins(q, e)

Deq()/Ok(e)
   requires ¬ isEmp(q)
   modifies at most (q)
   ensures q' = rest(q) ∧ e = first(q)
```

Fig. 1. Interfaces for queue operations.

predicate stands for the value of the object when the operation begins. A return formal or a primed argument formal, e.g., q', stands for the value of the object at the end of the operation. A **modifies at most** (*obj**) clause states that an operation may possibly change the value of any of the objects listed in *obj**; the values of objects not listed remain the same. The specification for *Deq* is *partial* since *Deq* is undefined for the empty queue.

As shown in Fig. 2, we use a *trait* written in the *Larch Shared Language* to describe the domain of values of a typed object. The set of operators and their signatures following **introduces** defines a vocabulary of terms to denote values. For example, *emp* and *ins*(*emp*, 5) denote two different queue values. The set of equations following the **asserts** clause defines a meaning for the terms, more precisely, an equivalence relation on the terms, and hence on the values they denote. For example, from *QVals*, we could prove that *rest*(*ins*(*ins*(*emp*, 3), 5)) = *ins*(*emp*, 5). The **generated by** clause of *QVals* asserts that *emp* and *ins* are sufficient operators to generate all values of queues. Formally, it introduces an inductive rule of inference

```
QVals: trait
 introduces
   emp: → Q
   ins: Q, E → Q
   first: Q → E
   rest: Q → Q
   isEmp: Q → Bool
 asserts
   Q generated by ( emp, ins )
   for all (q: Q, e: E)
      first(ins(q, e)) == if isEmp(q) then e else first(q)
      rest(ins(q, e)) == if isEmp(q) then emp else ins(rest(q), e)
      isEmp(emp) == true
      isEmp(ins(q, e)) == false
```

Fig. 2. Trait for queue values.

that allows one to prove properties of all terms of sort Q. We use the vocabulary of traits to write the assertions in the pre- and postconditions of a type's operations as described in the interface specifications; we use the meaning of equality to reason about its values.

2.2.2. *Legal Sequential Histories*

Since sequential specifications are sets of operation sequences and not sets of histories (event sequences), we need to draw a correspondence between histories and operation sequences.

Definition 5. If H is a sequential failure-free history, $OpSeq(H)$ is the operation sequence corresponding to H that is constructed as follows: For a transaction A, $OpSeq(H \mid A)$ is the operation sequence obtained from $H \mid A$ by pairing each invocation event with its matching response event, and by discarding commit events and pending invocation events. Let $A_1, \ldots, A_n$ be the transactions in H in the order in which they appear; then $OpSeq(H)$ is $OpSeq(H \mid A_1) \bullet \cdots \bullet OpSeq(H \mid A_n)$.

Note that the domain of $OpSeq$ is intentionally not restricted to complete histories. We define $OpSeq^*$ over a set of histories HS such that for each history H in HS. $OpSeq(H) \in OpSeq^*(HS)$.

Examples

If H_2 is the sequential failure-free history,

$$\begin{array}{lll} p & Enq(5) & A \\ p & Ok(\,) & A \\ p & Deq(\,) & A \\ p & Ok(5) & A \\ p & Commit & A \\ p & Enq(4) & B \\ p & Ok(\,) & B \\ p & Deq(\,) & B \end{array}$$

then $OpSeq(H_2)$ is the operation sequence:

$$\begin{array}{ll} p & Enq(5)/Ok(\,) \\ p & Deq(\,)/Ok(5) \\ p & Enq(4)/Ok(\,) \end{array}$$

Note that B's pending invocation is ignored.

End examples

Definition 6. A sequential failure-free history H is *legal* if for each object subhistory $H \mid x$, $OpSeq(H \mid x) \in x.sequential$.

A sequential history H involving multiple objects is *legal* if it is legal at each object, i.e., each subhistory $H \mid x$ is legal with respect to the sequential specification for x. If a sequential failure-free history $H \mid x$ is legal, we say $OpSeq(H \mid x)$ is *acceptable* (with respect to $x.sequential$); i.e., we use "legal" for event sequences and "acceptable" for operation sequences.

2.3. Atomicity = Serializability + Recoverability

We are interested in defining when a history is atomic, i.e., serializable and recoverable. We first define when a history is serializable and then (by constraining our histories of interest to only recoverable ones) when it is atomic.

Definition 7. If H is a history and T is a total order on transactions, $Seq(H, T)$ is a sequential history equivalent to H in which transactions appear in the order T.

For example, if $A_1, \ldots, A_n$ are transactions in H in the order T, then $Seq(H, T) = H \mid A_1 \bullet \cdots \bullet H \mid A_n$.

Serializability picks off only those equivalent sequential histories that are legal.

Definition 8. Let $S = committed(H) \cup active(H)$ in a history H. H is *serializable* if there exists some total order T on the transactions in S such that $Seq(H \mid S, T)$ is legal.

S is the set of transactions in H that have committed or are still active, and thus, does not include aborted transactions. By the above two definitions, serializability requires only that we find some total order T on non-aborted transactions in H that yields a legal sequential equivalent history.

Example

H_1 is serializable because ordering the transaction B before C is equivalent to the sequential history,

p	$Enq(2)$	B
p	$Ok(\,)$	B
q	$Enq(4)$	B
q	$Ok(\,)$	B
p	$Commit$	B
q	$Commit$	B
p	$Deq(\,)$	C
p	$Ok(2)$	C
p	$Enq(6)$	C
p	$Ok(\,)$	C

which is legal because C correctly dequeues 2, placed at the head of the queue by B. Notice that "equivalence" lets us ignore D because it has only a pending invocation in H_1 and "serializable" lets us ignore A because it aborts in H_1. Thus, we need only order B and C.

End example

A history H is *recoverable* if and only if all its transactions have committed, i.e., $aborted(H) = \varnothing$ and $active(H) = \varnothing$. To define when a history is atomic, we simply restrict S in Definition 8 to be just the set of committed transactions in H.

Definition 9. H is atomic if $H \mid committed(H)$ is serializable.

Recoverability lets us ignore noncommitted (i.e., aborted and active) transactions; a history is atomic if its recoverable subhistory is serializable. H_1 is atomic because it is equivalent to the sequential history that contains just the events of the one committed transaction (B) in H_1. Atomicity is a stronger property than serializability. For example, the following history,

$$
\begin{array}{lll}
q & Deq(\,) & A \\
q & Ok(3) & A \\
q & Commit & A \\
q & Ins(3) & B \\
q & Ok(\,) & B
\end{array}
$$

is not atomic because the recoverable subhistory is not equivalent to any legal sequential history for queues.

2.3.1. Local Atomicity

The only practical way to ensure atomicity in a decentralized distributed system is to have each object perform its own synchronization and recovery. In other words, we want to be able to verify the atomicity of a system composed of multiple objects by verifying the atomicity of individual objects.

However, atomicity as defined so far is too weak a property to let us perform such local reasoning. That is, H is not necessarily atomic just because $H \mid x$ is atomic for each object x. For example, suppose s and t are set objects with *Ins* and *Mem* operations, where *Ins* inserts an element into the set and *Mem* checks whether an element is a member of the set. The following history H_3 is not atomic, even though $H_3 \mid s$ and $H_3 \mid t$ both are:

s	$Ins(1)$	A
s	$Ok(\,)$	A
t	$Mem(2)$	A
t	$Ok(true)$	A
s	$Mem(1)$	B
s	$Ok(true)$	B
t	$Ins(2)$	B
t	$Ok(\,)$	B
s	$Commit$	A
s	$Commit$	B
t	$Commit$	A
t	$Commit$	B

$H_3|s$ is serializable in the order in which A precedes B and $H_3|t$ is serializable in the order in which B precedes A, but H_3 clearly cannot be serializable in an order consistent with both.

To ensure that all objects choose compatible serialization orders, it is necessary to impose certain additional restrictions on the behavior of atomic objects. These restrictions let us reason about atomicity locally. Thus, if each object is guaranteed to satisfy a local atomicity property, the entire system will be globally atomic. Avalon/C++ uses a local atomicity property that Weihl calls *hybrid atomicity.*[13]

Theorem 10. For all objects x in a system, if every history in $x.behavior$ is hybrid atomic, then every history in the system's behavior is atomic. [Weihl].

He also shows that hybrid atomicity is an *optimal* local atomicity property: no strictly weaker local property suffices to ensure global atomicity.[13]

Informally, a history H is *hybrid atomic* if it is serializable in the order in which the transactions in H commit. To capture formally the restriction that transactions must be serializable in commit-time order, we make the following adjustments to our model. When a transaction commits, it is assigned a logical timestamp,[21] which appears as an argument to that transaction's commit events. A commit event now looks like $x\ Commit(t)\ A$, where t is a timestamp. 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. For a given history H, let $TS(H)$ be the partial order such that $(A, B) \in TS(H)$ if A and B commit in H and the timestamp for A is less than the timestamp for B. $TS(H)$ defines a total order on $committed(H)$.

Definition 11. A history H is *hybrid atomic* if $H \mid committed(H)$ is serializable in the order $TS(H)$.

Serializability requires only that there exists some total order on transactions in H; atomicity implies we need order only the committed transactions; finally, hybrid atomicity picks an order (commit-time order) for which there must be a legal sequential equivalent.

Objects may learn of the commitment of transactions in an order different from the actual commit-time order. This behavior reflects real distributed systems where long delays or unreliable transmission of messages may cause objects not to have the most up-to-date view of the entire system. An object may not know that a committed transaction A has committed, and hence believe A is still active. The following history,

s	$Ins(1)$	A
s	$Ins(2)$	B
s	$Ok(\,)$	B
s	$Ok(\,)$	A
s	$Mem(2)$	A
s	$Ok(false)$	A
s	$Commit(1{:}15)$	B
s	$Commit(1{:}00)$	A

is hybrid atomic since it is serializable in the order in which A precedes B. Here, s learns about the commitment of A after it learns about the commitment of B, even though A commits before B.

Though all hybrid atomic histories are atomic, not all atomic histories are hybrid atomic.[13] Ignoring the timestamp arguments to the commit events, the following history,

s	$Ins(1)$	A
s	$Ok(\,)$	A
s	$Mem(1)$	B
s	$Ok(false)$	B
s	$Commit(1{:}00)$	A
s	$Commit(1{:}15)$	B

is atomic, but not hybrid atomic. It is serializable in the order in which B precedes A, but not in which A precedes B.

Since hybrid atomicity is local, we henceforth need only consider object subhistories.

2.3.2. On-line Atomicity

Since an object may hear about the commitment of transactions out-of-order, it may be difficult for it to choose an appropriate response to a pending invocation of an active transaction. Thus, we focus on "pessimistic" atomicity, where an active transaction with no pending invocation is always allowed to commit. Using this stronger property, called *on-line hybrid atomicity*, gives us the additional advantage that we can perform inductive reasoning over events in a history, which is not possible using simple hybrid atomicity.

Definition 12. H is *on-line atomic* if every well-formed history H' constructed by appending well-formed commit events to H is atomic. We call any sequential history equivalent to $H' \mid \text{committed}(H')$ a serialization of H.

This definition implies that H is on-line atomic if every one of its serializations is legal. We will typically work with serializations of H, letting us tack on zero, one, or more commit events to H. On-line atomicity allows us to choose to complete any number of active transactions, and thereby introduces nondeterminism into our correctness condition.

Examples

The following history,

q	$Enq(1)$	A
q	$Enq(2)$	B
q	$Ok(\,)$	B
q	$Ok(\,)$	A
q	$Commit(1:30)$	A
q	$Commit(1:15)$	B
q	$Deq(\,)$	C
q	$Ok(2)$	C

is on-line (hybrid) atomic. It has two serializations: one in which B precedes A, and C is not ordered with respect to A or B (in this case we assume C will abort); and one in which B precedes A and A precedes C (in this case we assume C will commit). It is easily verified that both are legal.

However, the following history, H_4,

q	$Enq(1)$	A
q	$Enq(2)$	B
q	$Ok(\,)$	B
q	$Ok(\,)$	A
q	$Commit(1:15)$	B
q	$Deq(\,)$	C
q	$Ok(2)$	C

is hybrid atomic but not on-line hybrid atomic, since the history $H'_4 = H_4 \bullet q\ Commit(1{:}00)A \bullet q\ Commit(1:30)C$ is not serializable in the order in which A precedes C. A may commit with a timestamp less than B's since A and B are concurrent, but C's timestamp must be greater than B's because it executes a response after B's commit.

End examples

3. VERIFICATION METHOD

We first define our notion of correctness based on the atomicity property presented in the previous section. We begin by defining what an implementation of an (abstract) object is in terms of histories.

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

Our correctness criterion for the implementation of an atomic object is defined below.

Definition 13. An implementation R for object a is *correct* if for every history H in R, $H \mid a$ is atomic.

We say object a is atomic; we typically do not require $H \mid r$ to be atomic. In a program, when the representation of an abstract object a is made up of multiple objects (e.g., an array plus two pointers) we treat the collection of objects as a single (representation) object.

Implementations and correctness are defined in terms of sets of histories; however, when verifying programs we need to reason about program state, and in particular, objects' values as they change from state to state. Hence, we need a method for relating histories to states, where informally a state can be viewed as a summary of a history. Then given program text, we make assertions about program state. Sometimes these assertions are as simple as saying something about an object's value after executing a program statement (e.g., $q' = f(q)$ for some function f); but, more generally, they are as complex as saying something about the set of possible histories that led up to a particular state. The method we give here is not the only way to prove correctness, but it is a way in which we have found useful when having to resolve the differences between a history-based model and a state-based piece of program text.

To show the correctness of an atomic object implementation, we must generalize techniques from the sequential domain. We use three "tools" in our method: (1) a representation invariant, (2) an abstraction function, and (3) the object's sequential specification. The representation invariant defines the domain of the abstraction function. The abstraction function maps a representation value to a set of sequences of abstract operations. The sequential specification determines which of those sequences are legal. The only unusual aspect of any of these tools is the range of the abstraction function: it is not a set of abstract values, but a set of sets of sequences of abstract operations.

Let *Rep* be the implementation type's set of values, *Abs* be the set of values of the (sequential) data type being implemented, and OP be the sequential object's set of operations. The subset of *Rep* values that are legal values is characterized by a predicate called the *representation invariant*, $\mathbf{I}$: $Rep \rightarrow bool$. The meaning of a legal representation is given by an *abstraction function*, $\mathbf{A}$: $Rep \rightarrow 2^{\text{OP}^*}$, defined only for values that satisfy the invariant. Unlike Hoare's abstraction functions for sequential objects[39] that map a representation value to a single abstract value, our abstraction functions map a representation value to a set of sequences of abstract operations.

Our basic verification method is to show inductively over events in a history that the following two properties are invariant. Let r be the representation value of the abstract object a in the state after accepting $OpSeq(H)$ for the history H, and let $Ser(H)$ denote the set of serializations of $H \mid a$.

1. $\mathbf{A}(r) \subseteq a.sequential$, and
2. $OpSeq^*(Ser(H)) \subseteq \mathbf{A}(r)$.

These two properties ensure that every serialization of H is a legal sequential history, and hence that H is on-line hybrid atomic. The first property says that $\mathbf{A}(r)$ contains only acceptable sequences of abstract operations. We explicitly appeal to the object's sequential specification to establish this property. The second property says that for each serialization in $Ser(H)$, the corresponding operation sequence is acceptable. In the inductive step of our proof technique, we show the invariance of these two properties across a history's events. That is, we check that after completing an operation that changes a history H to H' and r to r', $\mathbf{A}(r')$ still contains only acceptable operation sequences and for each serialization S in $Ser(H')S$ is legal, i.e., $OpSeq(S)$ is acceptable. This inductive verification method is specific for showing an object is on-line hybrid atomic, which by our definitions in the previous section suffices to show the more general correctness criterion of atomicity.

Note that if we were to replace the second property with the stronger requirement that $OpSeq^*(Ser(H)) = \mathbf{A}(r)$, then we could not verify certain correct implementations that keep track of equivalence classes of serializations. We give a specific example of this case in Section 5.

4. IMPLEMENTING ATOMIC OBJECTS

Given that atomicity is the fundamental correctness condition for objects in a transaction-based distributed system, how does one actually implement atomic objects? In this section we discuss some of the programming language support needed for constructing atomic objects. We have built this support in a programming language called Avalon/C++,[22] which is a set of extensions to C++.[23]

Essentially, Avalon/C++ provides ways to enable programmers to define abstract atomic types. For example, if we want to define an *atomic_array* type, we define a new class, *atomic_array*, which perhaps provides *fetch* and *store* operations. (Syntactically, a class is a collection of members, which are the components of the object's representation, and a collection of operation implementations.) The intuitive difference between a conventional *array* type and an *atomic_array* type is that objects of *array* type will not in general ensure serializability and recoverability of the transactions that access them whereas objects of *atomic_array* type will. However, the programmer who defines the abstract atomic type is still responsible for proving that the new type is correct, i.e., that all objects of the newly defined type are atomic. By providing language support for constructing atomic objects, we gain the advantage that this proof is done only once per class definition, not each time a new object is created. The verification method used for proving that an atomic type definition is correct is the heart of this paper.

Avalon/C++ has two built-in classes that together let programmers build atomic objects. The *trans_id* class provides operations that let programmers test the serialization order (i.e., commit-time order) of transactions at runtime. The *subatomic* class provides operations that let programmers ensure transaction serializability and recoverability.

4.1. Transaction Identifiers

The Avalon/C++ *trans_id* (transaction identifier) class provides ways for an object to determine the status of transactions at runtime, and thus synchronize the transactions that attempt to access it. *Trans_id*'s are a partially ordered set of unique values. Here is the *trans_id* class definition:

```
class trans_id {
    // ... internal representation omitted ....
public:
    trans_id();                          // constructor
    ~trans_id();                         // destructor
    trans_id&=(trans_id&)                // assignment
    bool operator==(trans_id&);          // equality
    bool operator<(trans_id&);           // serialized before?
    bool operator>(trans_id&);           // serialized after?
    bool done(trans_id&);                // committed to top level?
    friend bool descendant(trans_id&, trans_id&);
                    // is the first a descendant of the second?
    };
```

The three operations provided by *trans_id*'s relevant to this paper are the creation operation, the comparison operation <, and the *descendant* predicate.

The *creation* operation, called as follows:

$$trans\_id\ t = trans\_id(\ );$$

creates a new dummy subtransaction, commits it, and returns the subtransaction's *trans_id* to the parent transaction. Each call to the creation operation is guaranteed to return a unique *trans_id.* A *trans_id* is typically used as a "tag" on an operation. Calling the *trans_id* constructor allows a transaction to generate multiple *trans_id*'s ordered in the serialization order of the operations that created them.

The *comparison* operation, used in the following expression,

$$t1 < t2$$

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 $t2$ will also include the creation of $t1$, 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 *trans_id*'s may have the reverse ordering, or their ordering may be unknown.

Comparison induces a partial order on *trans_id*'s 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 *trans_id*'s will not become comparable to any new *trans_id*'s. Hence, "<" is capturing the commit-time order, i.e., serialization order, for committed transactions.

Finally, we use the *descendant* operation to compare whether a *trans_id* t' is a child of another. If the expression

$$descendant(t', t)$$

evaluates to true, then t' was created by the transaction t. Typically t is the *trans_id* of a committing or aborting transaction; the predicate lets us identify all its children. (A *friend* in C++ is a nonmember operation that is allowed access to the private part of a class.)

Avalon/C++ maintains a logically global *trans_id* tree that provides the information on the relationship among *trans_id*'s and the status of each transaction associated with a *trans_id*.

4.2. Ensuring Serializability and Recoverability

An atomic object in Avalon is defined by a C++ class that inherits from the Avalon built-in class *subatomic*. Here is the *subatomic* class definition:

```
class subatomic : public recoverable {
 protected:
   void seize();          // Gains short-term lock.
   void release();        // Releases short-term lock.
   void pause();          // Temporarily releases short-term lock.
 public:
 //... inherits two other operations from recoverable ...
   virtual void commit(trans_id& t);  // Called after transaction commit.
   virtual void abort(trans_id& t);   // Called after transaction abort.
};
```

A programmer defining a new atomic data type derives from class *subatomic*, gaining access to all the above operations. The details of each of these operations are not important to this paper. Roughly speaking, the first three operations permit the implementation of each of the operations of the user-defined atomic data type to be executed "indivisibly." This property is conventionally called "atomic," where atomicity is at the level of an individual operation (i.e., "all-or-nothing" of a single operation), as opposed to atomicity at the level of a transaction (i.e., "all-or-nothing" of a sequence of operations). The last two operations allow implementors control over the clean-up processing done by an object when it learns that a transaction has committed or aborted. Finally, for completeness, class *subatomic* itself inherits two operations from class *recoverable*. These two operations are used to ensure proper crash recovery, as distinct from transaction recovery; we do not address crash recovery in this paper.

With occasional minor variations, the implementation of each operation, *op*, of an atomic data type, *atomic_T*, which inherits from class *subatomic*, has the following form:

```
atomic_T::op(...) {
  trans_id t = trans_id();
  when(TEST)
      BODY;
}
```

As previously explained, the call to the creation operation of *trans_id* generates a new *trans_id* which is used to "tag" the current call to *op*. The *when* statement is a conditional critical region: BODY is executed only when TEST evaluates to *true*. Avalon/C++ implements the *when* statement in terms of the *seize*, *release*, and *pause* operations of *subatomic* and guarantees mutual exclusion at the operation level by associating a short-term lock with the object. TEST is typically an expression comparing (using *trans_id*'s "<" operation) *op*'s newly created *trans_id t* with other *trans_id*'s embedded in the object's representation. BODY typically computes a result and updates the object's state.

By inheriting from the *subatomic* class, the implementor can define new classes like *atomic_T*, and use the operations, in particular those encoded in the *when* statement, provided by *subatomic* to implement *atomic_T*'s operations. Since most operations follow the above template, the cleverness required in implementing operations of a new atomic type is in figuring out what the synchronization conditions on *atomic_T*'s operations are and then encoding a test for these conditions in each operation's TEST in order to maintain the commit-time order of transactions. [It remains an open research problem to figure out how to derive these synchronization conditions in a systematic way, as one does in computing weakest preconditions.] Proving correctness of the implementation focuses on showing that the synchronization conditions permit only atomic object histories.

Objects defined in a class that inherits from *subatomic* can also provide *commit* and *abort* operations that are called by the system as transactions commit or abort. A user-defined *commit* typically discards recovery information for the committing transaction, and a user-defined *abort* typically discards the tentative changes made by the aborting transaction. Intuitively, the execution of *commit* and *abort* operations in Avalon/C++ is 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. We do not address liveness properties in this paper, though certain ones are clearly of great interest. We could rely, for example, on the extensive work on temporal logic[24] for reasoning about liveness.

5. AN EXAMPLE: A HIGHLY CONCURRENT FIFO QUEUE

In this section, our verification technique is illustrated 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 enqueuing transactions, even though enqueuing operations 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 enqueuing and dequeuing transactions while the queue is non-empty.

We first give the Avalon/C++ implementation of the queue, then define the verification tools needed to prove its correctness, and then give a correctness proof.

5.1. The Implementation

As in the implementation of any abstract type, we present first the representation of the abstract type and then the implementations of each of the operations.

5.1.1. The Representation

We record information about *enq* operations in the following *struct*:

```
struct enq_rec {
  int item;                       // Item enqueued.
  trans_id enqr;                  // Who enqueued it.
  enq_rec(int i, trans_id& en)    // Constructor.
   {item = i; enqr = en;}
};
```

The *item* component is the enqueued item. The *enqr* component is a *trans_id* generated by the enqueuing transaction. We will use this *trans_id* to tag each *enq* operation. The last component defines a constructor operation for initializing the *struct*.

We record information about *deq* operations similarly:

```
struct deq_rec {
  int item;                      // Item dequeued.
  trans_id enqr;                 // Who enqueued it.
  trans_id deqr;                 // Who dequeued it.
  deq_rec(int i, trans_id& en, trans_id& de);  // Constructor.
   {item = i;
    enqr = en;
    deqr = de;
   }
};
```

The *deqr* component is a *trans_id* generated by the dequeuing transaction, which we will use to tag each *deq* operation. The constructor will be used to keep track of both the transaction that enqueued the item and the transaction that (tentatively) dequeues it.

We represent the queue as follows:

```
class atomic_queue : public subatomic {
  deq_stack deqd;              // Stack of dequeued items.
  enq_heap enqd;               // Heap of enqueued items.
 public:
  atomic_queue() {};           // Create empty queue.
  void enq(int item);          // Enqueue an item.
  int deq();                   // Dequeue an item.
  void commit(trans_id& t);    // Called on commit.
  void abort(trans_id& t);     // Called on abort.
};
```

The *deqd* component is a stack of *deq_rec*'s used to undo aborted *deq* operations. A stack provides *top*, *pop*, *push*, *is_empty*, and *clear* operations with the standard behaviors. The *enqd* component is a partially ordered heap of *enq_rec*'s, ordered by their *enqr* fields. A partially ordered heap provides the following operations: *insert*, to insert an *enq_rec*; *min_exists*, to test whether there exists a unique oldest *enq_rec*; *get_min*, to access it if it exists; *discard_min*, to remove it if it exists; and *discard*, to discard all *enq_rec*'s inserted by (aborted) transactions.

A typical scenario is that when an *enq* operation occurs, a new *trans_id* is generated and stored in a new *enq_rec*, along with the item being enqueued; this *enq_rec* is then inserted in the heap. When a *deq* operation occurs, a new *trans_id* is generated and stored in a new *deq_rec*, along with the information contained in the unique oldest *enq_rec* removed from the heap; this *deq_rec* is then pushed on the stack.

5.1.2. The Operations

If A is an active transaction, then we say B is *committed with respect to* A if B is committed, or if B and A are the same transaction. *Enq* and *deq* must satisfy the following synchronization constraints to ensure atomicity. Transaction A may dequeue an item if (1) the most recent transaction to have executed 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. The first condition ensures that A will not have dequeued the wrong item if the earlier dequeuer aborts, and the second condition ensures that there is something for A to dequeue. Similarly, A may enqueue an item if the last item dequeued was enqueued by a transaction committed with respect to A.

Given these conditions, here is the code for *enq*:

```
void atomic_queue::enq(int x) {
  trans_id tid = trans_id();
  when (deqd.is_empty() || (deqd.top()->enqr < tid))
    enqd.insert(x, tid);            // Record enqueue.
}
```

First, *Enq* creates a new *trans_id* to tag the execution of *enq* by the calling transaction. *Enq* then checks whether the item x most recently dequeued was enqueued by a transaction committed with respect to the caller. If so, the new *trans_id* and the new item are inserted in *enqd.* Otherwise, the transaction releases the short-term lock and tries again later (guaranteed by the implementation of the *when* statement). The somewhat complicated synchronization condition for *enq* is needed because transactions can perform multiple operations which must be ordered in the sequence in which they were called. (As an aside, the condition is also necessary and sufficient for nested transactions.) Consider the situation depicted in Fig. 3 (where time moves from left to right). A and B are two transactions where A performs two operations. Suppose A is still active. B must wait for A to commit because if B commits at an earlier time than A the second operation of A will have dequeued the wrong item (7, not 5, would be at the head of the queue).

Here is the code for *deq*:

```
int atomic_queue::deq() {
  trans_id tid = trans_id();
  when ( (deqd.is_empty() || deqd.top()->deqr < tid)
        && enqd.min_exists() && (enqd.get_min()->enqr < tid)) {
    enq_rec* min_er = enqd.delete_min();
    deq_rec dr(*min_er, tid);  // Move from enqueued heap...
    deqd.push(dr);             // to dequeued stack.
    return min_er->item;
  }
}
```

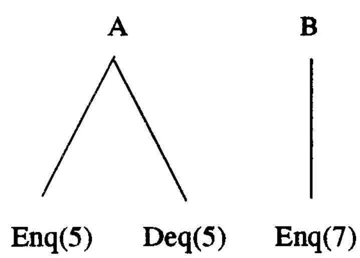

Fig. 3. An example of why an enqueuer (B) must wait.

Deq first creates a new *trans_id* to tag the execution of *deq* by the calling transaction. *Deq* then tests whether the most recent dequeuing transaction has committed with respect to the caller, and whether *enqd* has a unique oldest item. If the transaction that enqueued this item has committed with respect to the caller, it removes the item from *enqd* and records it in *deqd.* Otherwise, the caller releases the short-term lock, suspends execution, and tries again later. It is easy to see why a dequeuing transaction B must wait for the dequeuer A of the last dequeued item to be committed with respect to B. If B proceeds to dequeue without waiting for A to complete, then it will have dequeued the wrong item if A aborts. Consider the situation in Fig. 4 where 5 and 7 are the first and second elements in the queue. If A aborts then B should get a 5.

Note that an enqueuer does not have to wait for the dequeuer of the last dequeued item to commit. Consider the situation in Fig. 5. Suppose A has committed, but B has not. C can proceed to enqueue a 7 even though B has not yet completed. If B commits, it does not matter whether it commits before or after C. B will correctly see 5 at the head of the queue either way and C will correctly place 7 as the new head. If B aborts, then C will correctly place 7 after 5, which remains at the head of the queue. Thus, C can proceed without waiting for B to complete because there is no way C can be serialized before A and it does not matter in which order B and C are serialized.

In addition to the *enq* and *deq* operations, the *atomic_queue* provides *commit* and *abort* operations that are applied to the queue as transactions commit or abort. The *commit* operation looks like:

```
void atomic_queue::commit(trans_id& committer) {
  when (TRUE)                     // Always ok to commit.
    if (!deqd.is_empty() && descendant(deqd.top()->deqr, committer)) {
      deqd.clear();               // Discard all dequeue records.
    }
}
```

Recall that the runtime system calls individual objects' commit operations to inform objects of the commitment of some transaction. When the

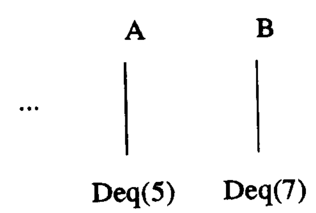

Fig. 4. An example of why a dequeuer (B) must wait.

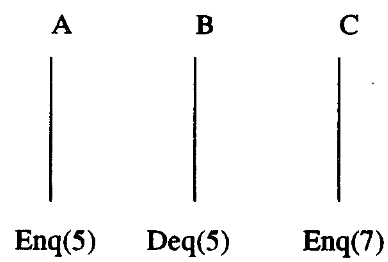

Fig. 5. An example of when an enqueuer (C) need not wait.

queue learns of the successful completion of *committer*, it can discard all *deq_rec*'s no longer needed for recovery. The implementation ensures that all *deq_rec*'s below the top are also superfluous, and can be discarded. We state this property formally when giving the representation invariant in Section 5.2.1.

The *abort* operation looks like:

```
void atomic_queue::abort(trans_id& aborter) {
  when (TRUE) {                 // Always ok to abort.
    while (!deqd.is_empty()     // Undo aborted dequeue by...
      && descendant(deqd.top()->deqr, aborter)) { // aborting transaction.
      deq_rec* d = deqd.pop(); // Undo aborted dequeue.
      enqd.insert(d->item, d->enqr);  // Put it back.
    }
    enqd.discard(aborter);      // Undo aborted enqueues.
  }
}
```

Similarly, when the runtime system calls *abort*, the object learns of the unsuccessful completion of some transaction (*aborter*). *Abort* undoes every operation executed by a transaction that is a descendant of the aborting transaction. It interprets *deqd* as an undo log, popping records for aborted operations, and inserting the items back in *enqd* heap. *Abort* then flushes all items enqueued by the aborted transaction and its descendants.

5.2. Application of Verification Method

As outlined in Section 3, we need to provide a representation invariant, abstraction function, and sequential specification in order to apply our verification method.

5.2.1. Representation Invariant

The queue operations preserve the following representation invariant. 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, For all representation values r:

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

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

2. Items are ordered in `deqd` by their enqueuing and dequeuing `trans_id`'s:

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

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

3. Any dequeued item must previously have been enqueued:

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

Thus, given an arbitrary state of the queue representation as in Fig. 6, where the stack grows upward: The first part of the representation invariant implies that x (and y) cannot be in any *enq_rec* in the heap. The second implies that $t1 < t2$ and $t1' < t2'$. The third implies that $t1 < t1'$ (and $t2 < t2'$).

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

5.2.2. Abstraction Function

Intuitively, the abstract value of the queue is defined in terms of what has been enqueued by committed transactions and may possibly be enqueued by active transactions (what is the heap) and what has been and may possibly be dequeued (what is on the stack). On-line atomicity

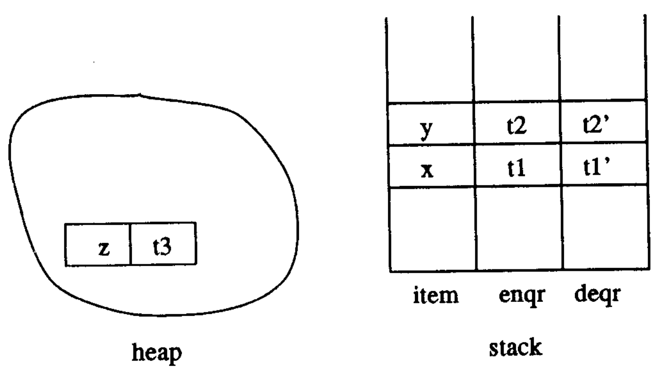

Fig. 6. An example queue representation state.

requires that we allow the possibility of active transactions to commit. For each, we pretend that it commits and reflect its effects—tentative enqueues and dequeues saved in the heap and stack—in the image of the abstraction function for a given representation value. When an active transaction actually does commit or the object finally finds out about the transaction's commitment, we know that we have already permitted for its effects to have taken place. Notice that both *commit* and *abort* do not change the abstract view of the queue, but only the representation.

To define the abstraction function, we need some auxiliary definitions. Let Q be a sequence of queue operations (not necessarily in *q.sequential*). Define the auxiliary functions $ENQ(Q)$ and $DEQ(Q)$ to yield the sequences of items enqueued and dequeued in Q:

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

For an operation p that is neither an $Enq(x)$ nor $Deq(x)$:

$$\mathrm{DEQ}(\mathrm{Q} \bullet \mathrm{p})) = \mathrm{DEQ}(\mathrm{Q}) \qquad \mathrm{ENQ}(\mathrm{Q} \bullet \mathrm{p}) = \mathrm{ENQ}(\mathrm{Q})$$

Here, "$Enq(x)$" is shorthand for an *Enq* operation "*q Enq*(x)/*Ok*( )" and "$Deq(x)$" is shorthand for "*q Deq*( )/*Ok*(x)," and "emp" denotes the empty sequence of items.

Let T be the universe of all (unique) *trans_id*'s. $P \subseteq T$ is a *prefix set* if, $\forall t, t' \in T$, if $t \in P \wedge t' < t$, then $t' \in P$. (The lemma below is independent of the queue example.)

Lemma 14. If H is an on-line atomic history for a set of *trans_id*'s and S is a serialization of H, then the *trans_id*'s whose creation operations appear in S form a prefix set.

Given a representation value r and a prefix set P of *trans_id*'s, we define the auxiliary function $OPS(r, P)$ to yield the partially ordered set of operations tagged by *trans_id*'s in P. $OPS(r, P)$ for the queue example is equal to the following set:

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

Each operation is tagged with a *trans_id* (*e.enqr*, *d.enqr*, or *d.deqr*). These *trans_id*'s induce a partial order on the elements of $OPS(r, P)$.

We define *Present'* to take a representation value r and a prefix set P, from which we can define $OPS(r, P)$. The value of *Present'*(r, P) is a set of sequences of (queue) operations where each sequence has the same elements as in $OPS(r, P)$ in a (total) order that extends the partial order of $OPS(r, P)$. Let $elems(S)$ be the set of elements in the sequence S and $< X$ denote the partial (total) order on elements in the partially (totally) ordered set X of elements.

$$Present'(r, P) = \{Q \mid OPS(r, P) = elems(Q) \wedge <_{OPS(r,P)} \subseteq <_{elems(Q)}\}$$

$Present(r)$ is defined as the union of $Present'(r, P)$ over all prefix sets $P \subseteq T$:

$$Present(r) = \bigcup_P Present'(r, P)$$

From the representation of the queue, we can see that a representation value r can keep track of only those items enqueued but not yet dequeued by committed transactions. In order to apply our verification method $(OpSeq^*(Ser(H)) \subseteq \mathbf{A}(r))$, we must consider all possible past histories that could have gotten the representation to its present state. So, we use *Past* to generate an infinite set of finite prefixes for a sequence of queue operations:

$$Past = \{Q \mid Q \in q.sequential \wedge DEQ(Q) = ENQ(Q)\}$$

Finally, we define $\mathbf{A}(r)$ as follows:

$$\mathbf{A}(r) = \{Q \mid \exists Q_1 \in Past, \exists Q_2 \in Present(r). Q = Q_1 \bullet Q_2\}$$

Note that $\mathbf{A}(r)$ typically includes more histories than $OpSeq^*(Ser(H))$ because of the many possible "equivalent" sequences of operations in *Past* that lead to the same current value of r.

5.2.3. Type-Specific Correctness Condition

The sequential queue specification in Figs. 1 and 2 defines the set of sequences of abstract queue operations, and thus, the set of legal abstract queue values. Our verification method requires that we reason about serializations of a given queue history. Recall we introduced serializations to capture the "pessimistic" property of on-line atomicity where a serialization of a history H is a sequential equivalent version of H with appended commit events for active transactions. Since an object may actually learn of the commitment of transactions in an order different from their actual commit times, we need a way to recognize when it is legal to insert (events of) an operation "in the middle" of a legal sequential history. The following lemma about sequences of queue operations captures this notion.

Lemma 15. Let $Q = Q_1 \bullet Q_2$ be an acceptable sequence of queue operations and let p be a queue operation. The sequence of queue operations $Q' = Q_1 \bullet p \bullet Q_2$ is acceptable if $DEQ(Q')$ is a prefix of $ENQ(Q')$.

This lemma captures the "prefix" property of a queue's behavior. It indirectly characterizes the synchronization conditions under which queue operations may execute concurrently.

To illustrate why we need to possibly reorder operations in a history, consider the following history:

q	$Enq(5)$	A
q	$Ok(\,)$	A
q	$Enq(7)$	A
q	$Ok(\,)$	A
q	$Deq(\,)$	A
q	$Ok(5)$	A
q	$Commit(1{:}00)$	A
q	$Enq(1)$	B
q	$Ok(\,)$	B
q	$Deq(\,)$	C
q	$Ok(7)$	C
q	$Enq(8)$	C
q	$Ok(\,)$	C

q must permit B and C to commit, and in either order. Though the sequence of operations that q sees is:

$$Enq(5) \bullet Enq(7) \bullet Deq(5) \bullet Enq(1) \bullet Deq(7) \bullet Enq(8)$$

since C may commit before B it needs to permit this sequence as well:

$$Enq(5) \bullet Enq(7) \bullet Deq(5) \bullet Deq(7) \bullet Enq(8) \bullet Enq(1)$$

where C's operations are inserted before B's. We need to check that both sequences are acceptable; and indeed, $\langle 5, 7 \rangle$ is a prefix of both $\langle 5, 7, 1, 8 \rangle$ and $\langle 5, 7, 8, 1 \rangle$.

5.3. Verifying the Implementation

C++, let alone Avalon/C++, does not have a formal semantics, so strictly speaking we cannot give a formal proof of correctness. Instead we assume standard meaning of imperative programming language constructs, e.g., sequential composition and procedure call, in the following proofs. Even were a formal semantics available for Avalon/C++, the proof outline would be similar to that presented here; in other words, we present the key insights that would be used in a formal proof.

We verify the queue implementation by showing inductively over events in histories that for every sequential history $S \in Ser(H)$ $OpSeq(S)$ lies in $\mathbf{A}(r)$ and that every sequence of queue operations in $\mathbf{A}(r)$ is acceptable. In our inductive step, we need consider only response, commit, and abort events, and not invocation events. Response events correspond to the completion of an operation, e.g., *Enq* or *Deq*. Next, we give a sketch of our arguments and then the more detailed proofs for the *Enq* and *Deq* operations.

5.3.1. Proof Sketch

Suppose the object completes an operation $Enq(x)$ with *trans_id* t, carrying the legal history H to H', and the representation r to r'. It is immediate from Lemma 14 that $OpSeq^*(Ser(H')) \subseteq \mathbf{A}(r')$. To show that every operation sequence in $\mathbf{A}(r')$ is acceptable, let $Q' \in \mathbf{A}(r')$. If Q' fails to satisfy the prefix property of Lemma 15, 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 *trans_id* for the item at the top of the *deqd* stack, and let t'' be the enqueuing *trans_id* for y. The *when* condition for *Enq* ensures that $t' < t$, and the representation invariant ensures that $t'' \leqslant t'$, hence that $t'' < t$, which is impossible if the *Enq* of x is serialized first.

Similarly, suppose the object completes an operation $Deq(x)$, carrying the representation r to r'. Let $Q = Q_1 \bullet Q_2 \in \mathbf{A}(r)$ and $Q' = Q_1 \bullet Deq(x) \bullet Q_2 \in \mathbf{A}(r')$. The representation 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 acceptable by Lemma 15.

If a *Commit* or *Abort* event carries a legal history H to H' and r to r', we must show that (1) $\mathbf{A}(r') \subseteq \mathbf{A}(r)$, and (2) no operation sequence in $\mathbf{A}(r) - \mathbf{A}(r')$ is in $OpSeq^*(Ser(H'))$. Property 1 ensures that every operation sequence in $\mathbf{A}(r')$ is acceptable, 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 S is not an element of $Ser(H')$, and hence $OpSeq(S)$ is not in $OpSeq^*(Ser(H'))$.

5.3.2. Detailed Proofs for Enqueue and Dequeue

In this section we will use induction to show the prefix property of Lemma 15. More specifically, if the prefix property holds of all operation sequences $s \in \mathbf{A}(r)$ at the invocation of the enqueue or dequeue operation, then it holds for all $s' \in \mathbf{A}(r')$ at the point of return. In the following, for $Q \in \mathbf{A}(r)$, $Q' \in \mathbf{A}(r')$, let $Q = Q_1 \bullet Q_2$ and $Q' = Q_1 \bullet op \bullet Q_2$ such that $\forall p \in Q_1 \neg(tid < trans\_id(p)) \wedge \forall p \in Q_2 \neg(trans\_id(p) < tid)$, where *op* is the enqueue or dequeue operation (as the case may be) with *trans_id tid*, and *trans_id*(*p*) is the *trans_id* of operation *p*.

Enqueue

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

```
void atomic_queue::enq(int x) {
  trans_id tid = trans_id();
  when (deqd.is_empty() || (deqd.top()->enqr < tid))

     WHEN: {∀y y ∈ elems(DEQ(s)) ⇒ trans_id(Enq(y)) < tid}

    enqd.insert(x, tid);

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

Proof. The following two Cases.

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 Q, which implies that the *trans_id* of the enqueue operation of y is ordered before *tid* by the WHEN assertion. The enqueue operation must be in Q_1 since (1) the *trans_id*'s of all enqueue operations of dequeued items are all ordered before that of *deqd.top*( ).*enqr* (by the representation invariant), which is ordered before *tid* (by the *when* condition); and (2) *tid* is not ordered before any operation in Q_1 (by the definition of $Q = Q_1 \bullet Q_2$). Since the enqueue operations of all dequeued items are in Q_1,

$$DEQ(Q)\ prefix\ ENQ(Q_1) \qquad (*)$$

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

$$\begin{aligned} DEQ(Q') = DEQ(Q), & \text{ which by } (*) \\ & \Rightarrow DEQ(Q')\ prefix\ ENQ(Q_1) \\ & \Rightarrow DEQ(Q')\ prefix\ ENQ(Q_1 \bullet e \bullet Q_2) \\ & \Rightarrow DEQ(Q')\ prefix\ ENQ(Q') \quad \blacksquare \end{aligned}$$

Dequeue

Here is the annotated *deq* operation:

```
int atomic_queue::deq() {
  trans_id tid = trans_id();
  when ((deqd.is_empty() || deqd.top()->deqr < tid)
        && enqd.min_exists() && (enqd.get_min()->enqr < tid)) {
```

```
    {WHEN: ∀ Deq operations d in s (trans_id(d) < tid ⇒ d in Q1)}
    enq_rec* min_er = enqd.delete_min(); // Transfer from enqueued heap...
    deq_rec dr(*min_er, tid);
    deqd.push(dr);                       // to dequeued stack.
    return min_er->item;
  }
    {POST: ∃ x. DEQ(s') = DEQ(s) • x ∧ ENQ(s') = ENQ(Q1) • ENQ(Q2)}
}
```

and the proof:

Proof. From the first conjunct of the *when* condition and the second clause of the representation invariant, we know that $DEQ(Q) = DEQ(Q_1)$. The second conjunct implies that there exists some $x = first(ENQ(Q) - DEQ(Q))$, 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 Q_1. Thus, by properties on sequences, there exists some $x = first(ENQ(Q_1) - DEQ(Q_1))$.

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

$$\begin{aligned} &DEQ(Q_1 \bullet d)\ prefix\ ENQ(Q_1 \bullet d) \\ &\Rightarrow DEQ(Q')\ prefix\ ENQ(Q_1 \bullet d) \\ &\Rightarrow DEQ(Q')\ prefix\ ENQ(Q_1 \bullet d \bullet Q_2) \\ &\Rightarrow DEQ(Q')\ prefix\ ENQ(Q') \end{aligned} \qquad \blacksquare$$

5.3.3. Summary of General Approach

Natually, this verification relies on properties of sequential queues. To verify an implementation of another data type, we would have to rely on a different set of properties, but the verification would follow a similar pattern:

- Define the representation invariant and abstraction function to capture how the set of possible serializations is encoded in the representation. In particular, here is a general technique for defining abstraction functions:
- Since operations are typically tagged by a *trans_id*, gather up these *trans_id*'s into a prefix set P, define a type-specific $OPS(r, P)$ (the elements are just individual instances of the operations provided by the type), define a *Past* function that generates all possible acceptable prefixes [Here again, an object's sequential specification determines how to define Past.] and finally a $Present'(r, P)$ function that turns a set of partially ordered operations into a set of totally ordered ones, i.e., sequences of operations.

- Define a type-specific analog to Lemma 15, capturing the basic synchronization conditions under which (events of) an operation can be inserted in the middle of a sequential history.
- Use an inductive argument to show that no operation, commit, or abort event can violate atomicity.

6. DISCUSSION AND RELATED WORK

6.1. Hybrid Atomicity Revisited

Atomicity has long been recognized as a basic correctness property within the database community.[25] More recently, several research projects have chosen atomicity as a useful foundation for general purpose distributed systems, including Argus,[2] Avalon,[3] Aeolus,[4] Camelot,[26] EXODUS[27] and Arjuna.[28] EXODUS and Arjuna, like Avalon, extend C++ to support recoverability, but neither gives programmers fine control over serializability. Of all these projects only Avalon and Argus provide linguistic support for programmers to design and implement user-defined atomic data types, which Weihl and Liskov argue is necessary for building large, realistic systems.[5]

One way to ensure atomicity of a set of concurrent transactions is to associate read and write locks with each object and to use a strict two-phase locking protocol to ensure serializability.[29] A transaction A obtains a read lock on an object x if A needs only to observe x's value. It obtains a write lock if it needs to update x's value. Each transaction first acquires all the locks it needs, then performs its operations on all objects for which it has obtained the appropriate locks, and then when it commits or aborts, releases all of its locks. Locks are held for the duration of a transaction, not individual operations; thus, two transactions that need to perform updates on the same object cannot proceed concurrently.

The two-phase read-write locking protocol is known to guarantee atomicity. However, since operations are naively divided into readers and writers, the amount of concurrency that can be obtained is restricted because the type semantics of objects are ignored. For example, consider the following two transactions that each perform two enqueue operations:

q	$Enq(1)$	A	q	$Enq(2)$	B
q	$Ok(\,)$	A	q	$Ok(\,)$	B
q	$Enq(3)$	A	q	$Enq(4)$	B
q	$Ok(\,)$	A	q	$Ok(\,)$	B

Following a read-write locking protocol would prevent A and B from executing concurrently. Suppose A has the write lock on q, then B

would not be able to obtain it, and hence has to wait for A to commit or abort before proceeding. Assuming both A and B commit, the only two permissible histories would both be sequential, where either all of A's operations precede all of B's or *vice versa*, i.e., A's and B's operations would not be interleaved. However, it should be possible to permit the following history in which A and B are executing concurrently:

q	$Enq(1)$	A
q	$Ok(\ )$	A
q	$Enq(2)$	B
q	$Ok(\ )$	B
q	$Enq(3)$	A
q	$Ok(\ )$	A
q	$Enq(4)$	B
q	$Ok(\ )$	B

Our correctness condition (hybrid atomicity) centainly permits this history since any extension of it with appended commit events for A and/or B is serializable.

Moreover, in the case when the queue is non-empty, we can permit a dequeuing transaction to proceed concurrently with an enqueuing one. Consider this example, which is a variation of the example drawn in Fig. 5:

q	$Enq(1)$	A
q	$Ok(\ )$	A
q	$Enq(3)$	A
q	$Ok(\ )$	A
q	$Commit(1{:}00)$	A
q	$Enq(2)$	B
q	$Deq(\ )$	C
q	$Ok(1)$	C
q	$Ok(\ )$	B
q	$Enq(4)$	B
q	$Ok(\ )$	B

The queue can permit C to perform a *Deq* operation and even return an element to C because it knows that A has committed and thus it knows what its first element is. Whether B commits or not, C still receives the correct element. Were two-phase read-write locking used, B and C would not be allowed to proceed concurrently because the *Enq* and *Deq* operations would both be classified as writers.

The use of commit-time serialization distinguishes Avalon from other transaction-based languages and systems, which are typically based on some form of strict two-phase locking. We chose to support commit-time serialization because it permits more concurrency than two-phase locking,[13] as well as better availability for replicated data.[30] Because commit-time serialization is compatible with strict two-phase locking, applications that use locking can still be implemented in Avalon/C++. In fact, we optimize for this more traditional case: As an alternative way to build atomic data types, programmers can inherit from another built-in Avalon/C++ called *atomic*, which provides access to read and write locks.

To summarize the results of this discussion and that in Section 2, the Venn diagram in Fig. 7 shows the relationship between atomic, hybrid atomic, and "two-phase-locking" atomic histories. Every "two-phase locking" history is hybrid atomic, but not conversely; every hybrid atomic history is atomic, but not conversely. The key point of this section is that hybrid atomicity provides more concurrency than "two-phase locking." The key point with respect to this paper, however, is that hybrid atomicity is local, whereas atomicity is not.

6.2. Other Models of Transactions

Lynch and Merritt,[12] and Best and Randell,[31] 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. 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.[20,32,33] None of these models were intended for reasoning about individual programs. Moreover, none are suitable for reasoning about high-level programming language constructs that include support for user-defined abstract data types.

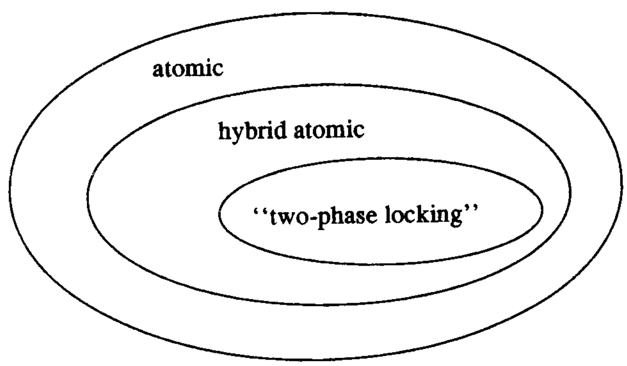

Fig. 7. Relationships among atomicity properties.

6.3. Abstraction Functions Revisited

The main contribution of this paper is the verification method used for showing the correctness of the implementation of an atomic data type. This method hinges on defining an abstraction function between a low-level view of an object and an abstract view. In the sequential domain, the signature of the abstraction function is:

$$\mathbf{A}\colon Rep \rightarrow Abs$$

Because of the on-line property of our correctness condition, in particular the inherent nondeterminism, we need to map to a set of values. A first attempt at extending the abstraction function would be to extend the range as follows:

$$\mathbf{A}\colon Rep \rightarrow 2^{Abs}$$

This extension is similar to Lynch and Tuttle's use of multi-valued *possibilities mappings*, where each "concrete state" maps to a set of "abstract states."[20] Lynch and her colleagues have used possibilities mappings to prove a wide range of distributed algorithms correct, including transaction-based locking and timestamp protocols. This abstraction function extension is also similar to what Herlihy and Wing needed to use in order to prove the correctness of *linearizable* objects,[7] again because of the inherent nondeterminism in the definition of the correctness condition.

However, even mapping to the powerset of abstract values is insufficient for the on-line hybrid atomic correctness condition we require. We need to keep track of sequences of operations because we need to permit reordering of operations. In fact, we need to be able to insert not just single operations into the middle of a history, but sequences of operations since a transaction may perform more than one operation. Hence, we finally extend the abstraction function's range to be:

$$\mathbf{A}\colon Rep \rightarrow 2^{OP^*}$$

In either extension, the standard trick of using auxiliary variables (Abadi and Lamport classify these into history and prophecy variables[34]) would also work. These variables can be included in the domain of the abstraction function (encoded as part of the representation state) and used (1) to keep track of the set of possible abstract values; (2) to log the history of abstract operations performed on the object so far; and (3) to keep track of implicit global data like the *trans_id* tree. Hence our abstraction functions can be turned into Abadi and Lamport's *refinement mappings*, where the extended domain of the representation state maps to a single abstract state.

In our initial approach to verification we tried to stick to a purely axiomatic approach in our verification method where we relied on Hoare-like axioms to reason about program statements, invariant assertions to reason about local state changes and global invariants, and auxiliary variables to record 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 events of each new operation (or operation sequences) are inserted somewhere "in the middle" of certain serializations. 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. Though the proofs given in this paper fall short of a pure syntax-directed verification, they could be completely axiomatized by encoding the set of serializations as auxiliary data. Even so, we have found that the resulting invariant assertions are syntactically intimidating and the proofs unintuitive and unnatural.

7. CURRENT AND FUTURE WORK

We have applied our verification method to a *directory* atomic data type, whose behavior is much more complex than a queue's. A directory stores key-item pairs and provides operations to insert, remove, alter, and lookup items given a key. Synchronization is done per key so transactions operating on different keys can execute concurrently. Moreover, we use an operation's arguments and results to permit, in some cases, operations on the same key to proceed concurrently. For example, an "unsuccessful" insertion operation, i.e., $Ins(k, x)/Ok(false)$, does not modify k's binding. so it does not conflict with a "successful" lookup operation, i.e., $Lookup(k)/Ok(y)$. Our proof of correctness relies on inductively showing a type-specific correctness condition, analogous to the "prefix" property for the queue. Informally stated, this condition says that it is legal to perform a successful remove, alter, lookup, or unsuccessful lookup, or insert on a key k as long as a key-item pair has already been successfully inserted for k and not yet successfully removed. This condition should hint to the reader that, in general, we need to keep track of the exact operations (including their arguments and results) and the order they have occurred already in a history to know which permutations are legal. We implemented the directory example in Avalon/C++.

In line with our philosophy of performing syntax-directed verification, we have used machine aids to verify the queue example. In particular, we used the Larch Shared Language to specify completely the queue representation, the set of queue abstract values (in terms of sequences of queue operations), the representation invariant, and the abstraction function. We used the Larch Prover[35] to prove the representation invariant holds of the representation and to perform the inductive reasoning we carried out in our verification method. We describe this work in more detail in Ref. 36.

Our work on specifying and verifying atomic data types and more recently, our work on using machine aids, has led us to explore extensions to our specification language. Two kinds of extensions seem necessary. First, we need a way to specify precisely and formally the synchronization conditions placed on each operation of an atomic object. We propose using a **when** clause analogous to the *when* statement found in Avalon/C++ or then WHEN assertion found in our proofs. Birrell *et al.* use informally a **when** clause in the Larch interface specification of Modula/2+'s synchronization primitives[37] and Lerner uses it to specify the queue example.[38] Second, we would like to extend the assertion language of the Larch interface specifications. The Larch Shared Language and the input language of the Larch Prover are both restricted to a subset of first-order logic. The assertions we write, however, in both the PRE/WHEN/POST conditions in our proofs and the **requires/when/ensures** clauses in our Larch interfaces, refer to operations, histories, sets of histories, and transactions directly, thereby requiring a richer and more expressive language than that which either the Larch Shared Language or the Larch Prover supports. Of course, for doing machine-aided verification with existing tools, we then would trade expressive power for computational tractability.

ACKNOWLEDGMENTS

I would like to thank Maurice Herlihy specifically for his close collaboration on the Avalon/C++ design and implementation, for his ideas about verifying atomicity, and for co-authoring a paper that presents a preliminary version of some of the ideas in this paper. I thank Bill Weihl for the original definitions of his model of transactions and atomicity properties.

I thank the participants of the 1989 REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness that took place in Plasmolen, The Netherlands. I especially thank Willem P. de Roever, Jr. who has been supportive and encouraging throughout my attempts at explaining this material.

I am grateful to the following members of the Avalon Project for reading drafts of this paper: David Detlefs, Chun Gong, Karen Kietzke, Linda Leibengood, Rick Lerner, Greg Morrisett, and Scott Nettles. Gong and Dave have been particularly helpful with some of the technical points.

Dave and Karen have been instrumental in turning the design of Avalon/C++ into a working implementation.

Finally, I thank the anonymous referees for their useful comments on improving this paper's presentation.

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 F33615-87-C-1499. Additional support was provided in part by the National Science Foundation under grant CCR-8620027. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

REFERENCES

1. C. H. Papadimitriou, The serializability of concurrent database updates, *Journal of the ACM* **26**(4):631–653 (October 1979).
2. B. H. Liskov and R. Scheifler, Guardians and actions: linguistic support for robust, distributed programs, *ACM Transactions on Programming Languages and Systems* **5**(3):381–404 (July 1983).
3. M. P. Herlihy and J. M. Wing, Avalon: Language Support for Reliable Distributed Systems, in *The 17th International Symposium on Fault-Tolerant Computing*, pp. 89–94 (July 1987). [Also available as CMU-CS-TR-86-167.]
4. 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 (December 1986).
5. 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).
6. K. R. Apt, N. Francez, and W. P. de Roever, A Proof System for Communicating Sequential Processes, *ACM Transactions on Programming Languages and Systems* **2**(3):359–385 (July 1980).
7. M. P. Herlihy and J. M. Wing, Axioms for concurrent objects, In *14th ACM Symposium on Principles of Programming Languages*, pp. 13–26 (January 1987).
8. L. Lamport, Specifying Concurrent Program Modules, *ACM Transactions on Programming Languages and Systems* **5**(2):190–222 (April 1983).
9. S. Owicki and D. Gries, Verifying Properties of Parallel Programs: An Axiomatic Approach, *Communications of the ACM* **19**(5):279–285 (May 1976).
10. J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Technical Report MIT/LCS/TR-260, Laboratory for Computer Science (April 1981).
11. D. P. Reed, Implementing atomic actions on decentralized data, *ACM Transactions on Computer Systems* **1**(1):3–23 (February 1983).
12. 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.]
13. W. E. Weihl, Specification and Implementation of Atomic Data Types, Technical Report TR-314, MIT Laboratory for Computer Science (March 1984).

14. W. E. Weihl, Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types, *Transactions on Programming Languages and Systems* **11**(2):249–283 (April 1989).
15. B. Nelson, *Remote Procedure Call*, Technical Report CMU-CS-81-119, Carnegie Mellon (May 1981).
16. J. Goree, *Internal Consistency of a Distributed Transaction System with Orphan Detection*, Technical Report MIT/LCS/TR-286, Laboratory for Computer Science (January 1983).
17. J. V. Guttag, J. J. Horning, and J. M. Wing, The Larch Family of Specification Languages, *IEEE Software* **2**(5):24–36 (September 1985).
18. D. Bjorner and C. G. Jones (Eds.), *Lecture Notes in Computer Science.* Volume 61: *The Vienna Development Method: the Meta-language*, Springer-Verlag, Berlin-Heidelberg-New York (1978).
19. J. M. Spivey, *Introducing Z: a Specification Language and its Formal Semantics*, Cambridge University Press (1988).
20. N. Lynch and M. Tuttle, *Hierarchical Correctness Proofs for Distributed Algorithms*, Technical Report MIT/LCS/TR-387, Laboratory for Computer Science (1987).
21. L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM* **21**(7):558–565 (July 1978).
22. D. L. Detlefs, M. P. Herlihy, and J. M. Wing, Inheritance of Synchronization and Recovery Properties in Avalon/C++, *IEEE Computer*:57–69 (December 1988).
23. B. Stroustrup, *The C++ Programming Language*, Addison Wesley (1986).
24. Z. Manna and A. Pnueli, Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs, *Science of Computer Programming* **4**:257–289 (1984).
25. 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).
26. A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, D. Duchamp, J. L. Eppinger, S. G. Menees, and D. S. Thompson, The Camelot Project, *Database Engineering* **9**(4) (December 1986). Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University (November 1986).
27. J. E. Richardson and M. J. Carey, Programming Constructs for Database System Implementation in EXODUS, In *ACM SIGMOD 1987 Annual Conference*, pp. 208–219 (May 1987).
28. G. Dixon and S. K. Shrivastava, Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems, In *Proc. of the 6th Symp. in Reliability in Distributed Software and Database Systems* (March 1987).
29. 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).
30. M. P. Herlihy, A quorum-consensus replication method for abstract data types, *ACM Transactions on Computer Systems* **4**(1):32–53 (February 1986).
31. E. Best and B. Randell, A Formal Model of Atomicity in Asynchronous Systems, *Acta Informatica* **16**(1):93–124 (1981).
32. 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.]
33. N. Lynch, *A Concurrency Control for Resilient Nested Transactions*, Technical Report MIT/LCS/TR-285, Laboratory for Computer Science (1985).
34. M. Abadi and L. Lamport, *The Existence of Refinement Mappings*, Technical Report 29, DEC Systems Research Center (August 1988).

35. S. J. Garland and J. V. Guttag, An Overview of LP, the Larch Prover, In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pp. 137–151. Chapel Hill, NC (April 1989). [Lecture Notes in Computer Science 355.]
36. J. M. Wing and C. Gong, Machine-Assisted Proofs of Properties of Avalon Programs, Technical Report CMU-CS-89-171 (August 1989).
37. A. Birrell, J. Guttag, J. Horning, R. Levin, Synchronization Primitives for a Multiprocessor: A Formal Specification. In *Proc. of the Eleventh ACM Symposium on Operating Systems Principles, ACM SIGOPS*, pp. 94–102 (1987).
38. R. A. Lerner, Specifying Concurrent Programs, Thesis Proposal (1989).
39. C. A. R. Hoare, Proof of Correctness of Data Representations, *Acta Informatica* **1**(1):271–281 (1972).

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium