Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

1. INTRODUCTION

1.1 Overview

Informally, a concurrent system consists of a collection of sequential processes that communicate through shared typed objects. This model encompasses both message-passing architectures in which the shared objects are message queues,

A preliminary version of this paper appeared in the Proceedings of the 14th ACM Symposium on Principles of Programming Languages, January 1987 [21].
This research was sponsored by IBM and the Defense Advanced Research Projects Agents (DOD), ARPA order 4976 (Amendment 20), under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB. Additional spport for J. M. Wing 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 authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.
Authors' address: Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

and shared-memory architectures in which the shared objects are data structures in memory. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to create and manipulate that object. In a sequential system, where an object's operations are invoked one at a time by a single process, the meaning of the operations can be given by pre- and postconditions. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to possible interleavings of operation invocations.

A concurrent computation is *linearizable* if it is "equivalent," in a sense formally defined in Section 2, to a legal sequential computation. We interpret a data type's (sequential) axiomatic specification as permitting only linearizable interleavings. Instead of leaving data uninterpreted, linearizability exploits the semantics of abstract data types; it permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using standard verification techniques. Unlike alternative correctness conditions such as sequential consistency [31] or serializability [40], linearizability is a *local* property: a system is linearizable if each individual object is linearizable. Locality enhances modularity and concurrency, since objects can be implemented and verified independently, and run-time scheduling can be completely decentralized. Linearizability is also a *nonblocking* property: processes invoking totally-defined operations are never forced to wait. Nonblocking enhances concurrency and implies that linearizability is an appropriate condition for systems for which real-time response is critical. Linearizability is a simple and intuitively appealing correctness condition that generalizes and unifies a number of correctness conditions both implicit and explicit in the literature.

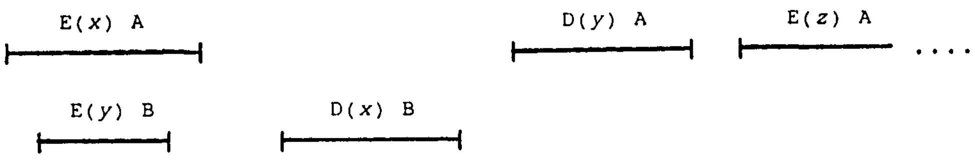
Using axiomatic specifications and our notion of linearizability, we can reason about two kinds of problems:

(1) We reason about the correctness of linearizable object implementations using new techniques that generalize the notions of representation invariant and abstraction function [18, 25] to the concurrent domain.
(2) We reason about computations that use linearizable objects by transforming assertions about concurrent computations into simpler assertions about their sequential counterparts.

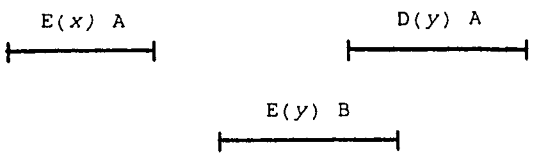
Section 2 presents our model of a concurrent system and the formal definition of linearizability. Section 3 discusses linearizability's locality and nonblocking properties and compares it to other correctness conditions. Section 4 presents our proof technique for reasoning about implementations of linearizable objects, and illustrates this technique on two novel implementations of a highly concurrent queue. Section 5 presents examples of reasoning about concurrent registers and queues, given that they are linearizable. Section 6 surveys some related work and discusses the significance of linearizability as a correctness condition.

1.2 Motivation

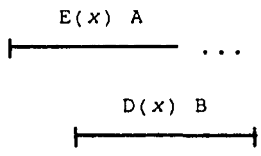
When defining a correctness condition for concurrent objects, two requirements seem to make intuitive sense: First, each operation should appear to "take effect" instantaneously, and second, the order of nonconcurrent operations should be



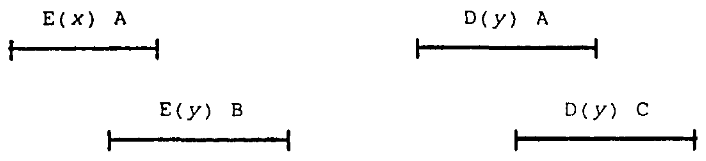
(a) H_1 (acceptable).



(b) H_2 (not acceptable).



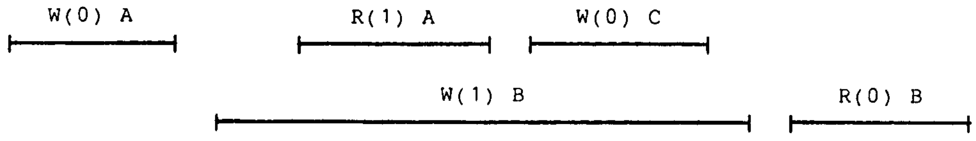
(c) H_3 (acceptable).



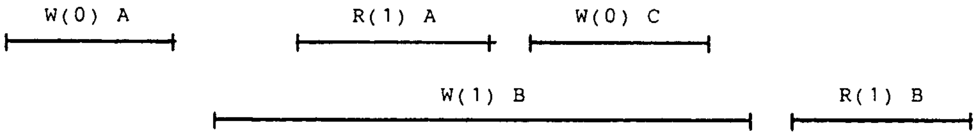
(d) H_4 (not acceptable).

Fig. 1. FIFO queue histories.

preserved. These requirements allow us to describe acceptable concurrent behavior directly in terms of acceptable sequential behavior, an approach that simplifies both formal and informal reasoning about concurrent programs. We capture these notions formally in the next section; here we informally review some examples to illustrate what we do and do not consider intuitively acceptable concurrent behavior. Our examples employ a first in, first out (FIFO) queue, a simple data type that provides two operations: *Enq* inserts an item in the queue, and *Deq* returns and removes the oldest item from the queue. Figure 1 shows four different ways in which a FIFO queue might behave when manipulated by concurrent processes. Here, a time axis runs from left to right, and each operation



(a) H_5 (acceptable).



(b) H_6 (not acceptable).

Fig. 2. Register histories.

is associated with an interval. Overlapping intervals indicate concurrent operations. We use "E(x) A" ("D(x) A") to stand for the enqueue (dequeue) operation of item x by process A.

The behavior shown in H_1 (Figure 1a) corresponds to our intuitive notion of how a concurrent FIFO queue should behave. In this scenario, processes A and B concurrently enqueue x and y. Later, B dequeues x, and then A dequeues y and begins enqueuing z. Since the dequeue for x precedes the dequeue for y, the FIFO property implies that their enqueues must have taken effect in the same order. In fact, their enqueues were concurrent, thus they could indeed have taken effect in that order. The uncompleted enqueue of z by A illustrates that we are interested in behaviors in which processes are continually executing operations, perhaps forever.

The behavior shown in H_2, however, is not intuitively acceptable. Here, it is clear to an external observer that x was enqueued before y, yet y is dequeued without x having been dequeued. To be consistent with our informal requirements, A should have dequeued x. We consider the behavior shown in H_3 to be acceptable, even though x is dequeued before its enqueuing operation has returned. Intuitively, the enqueue of x took effect before it completed. Finally, H_4 is clearly unacceptable because y is dequeued twice.

To decide whether a concurrent history is acceptable, it is necessary to take into account the object's intended semantics. For example, acceptable concurrent behaviors for FIFO queues would not be acceptable for stacks, sets, directories, etc. When restricted to register objects providing read and write operations, our intuitive notion of acceptability corresponds exactly to the notion used in Misra's careful axiomatization of concurrent registers [35]. Our approach can be thought of as generalizing Misra's approach to objects with richer sets of operations. For example, H_5 in Figure 2a is acceptable, but H_6 is not (examples are taken from [35]). These two behaviors differ at one point: In H_5, B reads a 0, and in H_6, B reads a 1. The latter is intuitively unacceptable because A did a previous read of a 1, implying that B's write of 1 must have occurred before A's read. C's subsequent write of 0, though concurrent with B's write of 1, strictly follows A's read of 1.

In the next section, we formalize the intuition presented here by defining the notion of linearizability to encompass those histories we have argued are intuitively acceptable.

2. SYSTEM MODEL AND DEFINITION OF LINEARIZABILITY

2.1 Histories

Informally, a concurrent system consists of a collection of sequential threads of control called *processes* that communicate through shared data structures called *objects*. Each object has a unique *name* and a *type*. The type defines a set of possible *values*, and a set of primitive *operations* that provide the only means to manipulate that object. Processes are sequential: each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response. (Dynamic process creation can be modeled simply by treating each child process as an additional process that executes no operations before the *fork* or after the *join*.)

Formally, an execution of a concurrent system is modeled by a *history*, which is a finite sequence of operation *invocation* and *response* events. A *subhistory* of a history H is a subsequence of the events of H. An operation invocation is written as $\langle x\ op(args^*)\ A\rangle$, where x is an object name, *op* is an operation name, *args** denotes a sequence of argument values, and A is a process name. The response to an operation invocation is written as $\langle x\ term(res^*)\ A\rangle$, where *term* is a termination condition, and *res** is a sequence of results. We use "Ok" for normal termination. A response *matches* an invocation if their object names agree and their process names agree. An invocation is *pending* in a history if no matching response follows the invocation. If H is a history, *complete*(*H*) is the maximal subsequence of H consisting only of invocations and matching responses.

A history H is *sequential* if:

(1) The first event of H is an invocation.

(2) Each invocation, except possibly the last, is immediately followed by a matching response. Each response is immediately followed by a matching invocation.

A history that is not sequential is *concurrent.*

A *process subhistory*, H | P (H at P), of a history H is the subsequence of all events in H whose process names are P. An object subhistory H | x is similarly defined for an object x. Two histories H and H′ are *equivalent* if for every process P, H | P = H′ | P. A history H is *well-formed* if each process subhistory H | P of H is sequential. All histories considered in this paper are assumed to be well-formed. Notice that whereas process subhistories of a well-formed history are necessarily sequential, object subhistories are not.

An *operation*, e, in a history is a pair consisting of an invocation, $inv(e)$, and the next matching response, $res(e)$. We denote an operation by $[q\ inv/res\ A]$, where q is an object and A a process. An operation e_0 *lies within* another operation e_1 in H if $inv(e_1)$ precedes $inv(e_0)$ and $res(e_0)$ precedes $res(e_1)$ in H. Angle brackets for events and square brackets for operations are omitted where they would otherwise be unnecessarily confusing; object and process names are omitted where they are clear from context.

For example, H_1 of Figure 1 is the following well-formed history for a FIFO queue q.

```
q Enq(x) A
q Enq(y) B
q Ok( ) B
q Ok( ) A
q Deq( ) B
q Ok(x) B
q Deq( ) A
q Ok(y) A
q Enq(z) A
```

The first event in H_1 is an invocation of Enq with argument x by process A, and the fourth event is the matching response with termination condition Ok and no results. The [q Enq(y)/Ok( ) B] operation lies within the [q Enq(x)/Ok( ) A] operation. The subhistory, complete (H_1), is H_1 with the last (pending) invocation of Enq removed. Reordering the first two events yields one of many histories equivalent to H_1.

A set S of histories is *prefix-closed* if, whenever H is in S, every prefix of H is also in S. A *single-object* history is one in which all events are associated with the same object. A *sequential specification* for an object is a prefix-closed set of single-object sequential histories for that object. A sequential history H is *legal* if each object subhistory $H \mid x$ belongs to the sequential specification for x. Many conventional techniques exist for defining sequential specifications. In this paper, we use the axiomatic style of Larch [19], in which an object's sequential history is summarized by a *value*, which (informally speaking) reflects the object's state at the end of the history. These values are used in axioms giving the pre- and postconditions on the objects operations. For example, axioms for the Enq and Deq operations for FIFO queues are shown in Figure 3. The post-condition for Enq states that on termination, the new queue value is the old queue value with e inserted. The specification for Deq states that applying that operation to a non-empty queue removes the first item from the queue. An operation is *total* if, like Enq, it is defined for every object value, otherwise it is *partial*, like Deq which is left undefined for the empty queue.

2.2 Definition of Linearizability

A history H induces an irreflexive partial order $<_H$ on operations:

$$e_0 <_H e_1 \text{ if res}(e_0) \text{ precedes inv}(e_1) \text{ in H.}$$

Axiom E:

$$\{\text{true}\}$$
$$\text{Enq}(e)\ /\ \text{Ok}()$$
$$\{q' = \text{ins}(q, e)\}$$

Axiom D:

$$\{q \neq []\}$$
$$\text{Deq}()\ /\ \text{Ok}(e)$$
$$\{q' = \text{rest}(q) \land e = \text{first}(q)\}$$

Fig. 3. Axioms for queue operations.

(Where appropriate, subscripts on partial orders are omitted). Informally, $<_H$ captures the "real-time" precedence ordering of operations in H. Operations unrelated by $<_H$ are said to be *concurrent.* If H is sequential, $<_H$ is a total order.

A history H is *linearizable* if it can be extended (by appending zero or more response events) to some history H′ such that:

L1: complete(H′) is equivalent to some legal sequential history S, and

L2: $<_H \subseteq <_S$.

Informally, extending H to H′ captures the notion that some pending invocations may have taken effect even though their responses have not yet been returned to the caller (as in the pending Enq in history H_3 in Figure 1). Restricting attention to complete(H′) captures the notion that the remaining pending invocations have not yet had an effect. L1 states that processes act as if they were interleaved at the granularity of complete operations. L2 states that this apparent sequential interleaving respects the real-time precedence ordering of operations.

We call S a *linearization* of H. Nondeterminism is inherent in the notion of linearizability: (1) For each H, there may be more than one extension H′ satisfying the two conditions, L1 and L2, and (2) for each extension H′, there may be more than one linearization S. A *linearizable object* is one whose concurrent histories are linearizable with respect to some sequential specification.

2.3 Queue Examples Revisited

Let "·" denote concatenation of events. The history H_1 shown in Figure 1 is linearizable, because $H_1 \cdot \langle q\ \text{Ok}(\ )\ \text{A}\rangle$ is equivalent to the following sequential history:

q Enq(x) A (History H_1')
q Ok( ) A
q Enq(y) B
q Ok( ) B
q Deq( ) B
q Ok(x) B
q Deq( ) A
q Ok(y) A
q Enq(z) A
q Ok( ) A

H_2 is not linearizable:

```
q Enq(x) A    (History H2)
q Ok( ) A
q Enq(y) B
q Deq( ) A
q Ok( ) B
q Ok(y) A
```

because the complete Enq operation of x precedes the Enq of y, but y is dequeued before x.

Linearizability does not rule out histories such as H_3, in which an operation "takes effect" before its return event occurs:

```
q Enq(x) A    (History H3)
q Deq( ) B
q Ok(x) B
```

H_3 can be extended to $H_3' = H_3 \cdot \langle q\ \text{Ok( ) A}\rangle$, which is equivalent to the sequential history in which the enqueue operation occurs before the dequeue.

Finally, H_4,

```
q Enq(x) A    (History H4)
q Enq(y) B
q Ok( ) A
q Ok( ) B
q Deq( ) A
q Deq( ) C
q Ok(y) A
q Ok(y) C
```

is not linearizable because y is enqueued once but dequeued twice, and hence H_4 is not equivalent to any sequential FIFO queue history.

3. PROPERTIES OF LINEARIZABILITY

This section proves that linearizability is a *local* and *nonblocking* property, and discusses the differences between it and other correctness conditions.

3.1 Locality

A property P of a concurrent system is said to be *local* if the system as a whole satisfies P whenever each individual object satisfies P. Linearizability is a *local* property:

THEOREM 1. *H is linearizable if and only if, for each object x, H | x is linearizable.*

PROOF. The "only if" part is obvious.

For each x, pick a linearization of $H \mid x$. Let R_x be the set of responses appended to $H \mid x$ to construct that linearization, and let $<_x$ be the corresponding linearization order. Let H' be the history constructed by appending to H each response in R_x. We will construct a partial order $<$ on the operations of complete(H') such that: (1) For each x, $<_x \subseteq <$, and (2) $<_H \subseteq <$. Let S be the sequential history constructed by ordering the operations of complete(H') in any total order that extends $<$. Condition (1) implies that S is legal, hence that L1 is satisfied, and Condition (2) implies that L2 is satisfied.

Let $<$ be the transitive closure of the union of all $<_x$ with $<_H$. It is immediate from the construction that $<$ satisfies Conditions (1) and (2), but it remains to be shown that $<$ is a partial order. We argue by contradiction. If not, then there exists a set of operations $e_1, \ldots, e_n$, such that $e_1 < e_2 < \cdots < e_n$, $e_n < e_1$, and each pair is directly related by some $<_x$ or by $<_H$. Choose a cycle whose length is minimal.

Suppose all operations are associated with the same object x. Since $<_x$ is a total order, there must exist two operations e_{i-1} and e_i such that $e_{i-1} <_H e_i$ and $e_i <_x e_{i-1}$, contradicting the linearizability of x.

The cycle must therefore include operations of at least two objects. By reindexing if necessary, let e_1 and e_2 be operations of distinct objects. Let x be the object associated with e_1. We claim that none of $e_2, \ldots, e_n$ can be an operation of x. The claim holds for e_2 by construction. Let e_i be the first operation in $e_3, \ldots, e_n$ associated with x. Since e_{i-1} and e_i are unrelated by $<_x$, they must be related by $<_H$; hence the response of e_{i-1} precedes the invocation of e_i. The invocation of e_2 precedes the response of e_{i-1}, since otherwise $e_{i-1} <_H e_2$, yielding the shorter cycle $e_2, \ldots, e_{i-1}$. Finally, the response of e_1 precedes the invocation of e_2, since $e_1 <_H e_2$ by construction. It follows that the response to e_1 precedes the invocation of e_i, hence $e_1 <_H e_i$, yielding the shorter cycle $e_1, e_i, \ldots, e_n$.

Since e_n is not an operation of x, but $e_n < e_1$, it follows that $e_n <_H e_1$. But $e_1 <_H e_2$ by construction, and because $<_H$ is transitive, $e_n <_H e_2$, yielding the shorter cycle $e_2, \ldots, e_n$, the final contradiction. □

Henceforth, we need consider only single-object histories.

Locality is important because it allows concurrent systems to be designed and constructed in a modular fashion; linearizable objects can be implemented, verified, and executed independently. A concurrent system based on a nonlocal correctness property must either rely on a centralized scheduler for all objects, or else satisfy additional constraints placed on objects to ensure that they follow compatible scheduling protocols. Locality should not be taken for granted; as discussed below, the literature includes proposals for alternative correctness properties that are not local.

3.2 Blocking versus Nonblocking

Linearizability is a *nonblocking* property: a pending invocation of a totally-defined operation is never required to wait for another pending invocation to complete.

THEOREM 2. *Let inv be an invocation of a total operation. If* $\langle x \text{ inv } P\rangle$ *is a pending invocation in a linearizable history H, then there exists a response* $\langle x \text{ res } P\rangle$ *such that* $H \cdot \langle x \text{ res } P\rangle$ *is linearizable.*

PROOF. Let S be any linearization of H. If S includes a response $\langle x \text{ res P}\rangle$ to $\langle x \text{ inv P}\rangle$, we are done, since S is also a linearization of $H \cdot \langle x \text{ res P}\rangle$. Otherwise, $\langle x \text{ inv P}\rangle$ does not appear in S either, since linearizations, by definition, include no pending invocations. Because the operation is total, there exists a response $\langle x \text{ res P}\rangle$ such that

$$S' = S \cdot \langle x \text{ inv P}\rangle \cdot \langle x \text{ res P}\rangle$$

is legal. S′, however, is a linearization of H · ⟨x res P⟩, and hence is also a linearization of H. □

This theorem implies that linearizability per se never forces a process with a pending invocation of a total operation to block. Of course, blocking (or even deadlock) may occur as artifacts of particular implementations of linearizability, but is is not inherent to the correctness property itself. (Techniques for constructing nonblocking implementations of linearizable objects are discussed elsewhere [23].) This theorem suggests that linearizability is an appropriate correctness condition for systems where concurrency and real-time response are important. We shall see that alternative correctness conditions, such as serializability, do not share this nonblocking property.

The nonblocking property does not rule out blocking in situations where it is explicitly intended. For example, it may be sensible for a process attempting to dequeue from an empty queue to block, waiting until another process enqueues an item. Our queue specification captures this intention by making Deq's specification partial, leaving it undefined for the empty queue. The most natural concurrent interpretation of a partial sequential specification is simply to wait until the object reaches a state in which the operation is defined.

3.3 Comparison to Other Correctness Conditions

Lamport's notion of *sequential consistency* [31] requires that a history be equivalent to a legal sequential history. Sequential consistency is weaker than linearizability, because it does not require the original history's precedence ordering to be preserved. For example, history H_7 is sequentially consistent, but not linearizable:

q Enq(x) A (History H_7)
q Ok( ) A
q Enq(y) B
q Ok( ) B
q Deq( ) B
q Ok(y) B

Sequential consistency is not a local property. Consider the following history H_8, in which processes A and B operate on queue objects p and q.

p Enq(x) A (History H_8)
p Ok( ) A
q Enq(y) B
q Ok( ) B
q Enq(x) A
q Ok( ) A
p Enq(y) B
p Ok( ) B
p Deq( ) A
p Ok(y) A
q Deq( ) B
q Ok(x) B

It is easily checked that $H_8 \mid p$ and $H_8 \mid q$ are sequentially consistent, but H_8 itself it not.

Much work on databases and distributed systems uses *serializability* [40] as the basic correctness condition for concurrent computations.[1] In this model, a *transaction* is a thread of control that applies a finite sequence of primitive operations to a set of objects shared with other transactions.[2] A history is *serializable* if it is equivalent to one in which transactions appear to execute sequentially, i.e., without interleaving. A (partial) precedence order can be defined on non-overlapping pairs of transactions in the obvious way. A history is *strictly serializable* if the transactions' order in the sequential history is compatible with their precedence order. Strict serializability is ensured by some synchronization mechanisms, such as two-phase locking [12], but not by others, such as multi-version timestamp schemes [41], or schemes that provide high levels of availability in the presence of network partitions [22].

Linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object. Nevertheless, this single-operation restriction has far-reaching practical and formal consequences, giving linearizable computations a different flavor from their serializable counterparts. An immediate practical consequence is that concurrency control mechanisms appropriate for serializability are typically inappropriate for linearizability because they introduce unnecessary overhead and place unnecessary restrictions on concurrency. For example, the queue implementation given below in Section 4 is much more efficient and much more concurrent than an analogous implementation using conventional serializability-oriented techniques such as two-phase locking or multi-version timestamping.

One important formal difference between linearizability and serializability is that neither serializability nor strict serializability is a local property. For example, in history H_8 shown above, if we interpret A and B as transactions instead of processes, then it is easily seen that both $H_8 \mid p$ and $H_8 \mid q$ are strictly serializable but H_8 is not. (Because A and B overlap at each object, they are unrelated by transaction precedence in either subhistory.) Moreover, since A and B each dequeues an item enqueued by the other, H_8 is not even serializable. A practical consequence of this observation is that implementors of objects in serializable systems must rely on global conventions to ensure that all objects' concurrency control mechanisms are compatible with one another. For example, it is well known that two-phase locking is incompatible with multiversion timestamping [46].

Another important formal difference is that serializability places more rigorous restrictions on concurrency. Serializability is inherently a *blocking* property: under certain circumstances, a transaction may be unable to complete a pending operation without violating serializability, even if the operation is total. Such a transaction must be rolled back and restarted, implying that additional mechanisms must be provided for that purpose. For example, consider the following

[1] In practice, serializability is almost always provided in conjunction with *failure atomicity*, ensuring that a transaction unable to execute to completion will be automatically rolled back. There is no counterpart to failure atomicity for linearizability.

[2] Some models permit transactions to be nested, or to encompass concurrent threads of control. Our remarks about locality and nonblocking hold for these more elaborate models as well.

history involving two register objects: x and y, and two transactions: A and B.

```
x Read( ) A     (History H9)
y Read( ) B
x Ok(0) A
y Ok(0) B
x Write(1) B
y Write(1) A
```

Here, A and B respectively read x and y and then attempt to write new values to y and x. It is easy to see that both pending invocations cannot be completed without violating serializability. Although different concurrency control mechanisms would resolve this conflict in different ways, such deadlocks are not an artifact of any particular mechanism; they are inherent to the notion of serializability itself. By contrast, we have seen that linearizability never forces processes executing total operations to wait for one another.

Perhaps the major practical distinction between serializability and linearizability is that the two notions are appropriate for different problem domains. Serializability is appropriate for systems such as databases in which it must be easy for application programmers to preserve complex application-specific invariants spanning multiple objects. A general-purpose serialization protocol, such as two-phase locking, enables programmers to reason about transactions as if they were sequential programs (setting aside questions of deadlock or performance). Linearizability, by contrast, is intended for applications such as multiprocessor operating systems in which concurrency is of primary interest, and where programmers are willing to apply special-purpose synchronization protocols, and to reason explicitly about the effects of concurrency.

4. VERIFYING THAT IMPLEMENTATIONS ARE LINEARIZABLE

In this section, we motivate and describe our method for verifying implementations of linearizable objects. We begin with our definition of when an implementation is correct. In order to prove correctness, we reexamine the notions of representation invariant and abstraction function (Section 4.2), and use their new interpretation in our proof method (Section 4.3).

4.1 Definition of Correctness

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

An implementation is *correct* with respect to the specification of ABS if for every history H in the implementation, H | ABS is linearizable.

4.2 Representation Invariant and Abstraction Function

We first review how to verify the correctness of sequential objects [18, 25]. In the sequential domain, an implementation consists of an *abstract type* ABS, the type being implemented, and a *representation* type REP, the type used to implement ABS. The subset of REP values that are *legal* representations is characterized by a predicate called the *rep invariant*, **I**: REP → BOOL. The meaning of a legal representation is given by an *abstraction function*, **A**: REP → ABS, defined for representation values that satisfy the invariant.

An abstract operation α is implemented by a sequence, ρ, of rep operations that carries the rep from one legal value to another, perhaps passing through intermediate values where the abstraction function is undefined. The rep invariant is thus part of both the precondition and postcondition for each operation's implementation; it must be satisfied between abstract operations, although it may be temporarily violated while an operation is in progress. An implementation, ρ, of an abstract operation, α, is *correct* if there exists a rep invariant, **I**, and abstraction function, **A**, such that whenever ρ carries one legal rep value r to another r', α carries the abstract value from $\mathbf{A}(r)$ to $\mathbf{A}(r')$.

This verification technique must be substantially modified before it can be applied to concurrent objects: we change both the meaning of the rep invariant and the signature of the abstraction function. To help motivate these changes and to make our discussion as concrete as possible, consider the following highly concurrent implementation of a linearizable FIFO queue. The queue's representation is a record with two components: *items* is an array having a low bound of 1 and a (conceptually) infinite high bound, and *back* is the (integer) index of the next unused position in *items*.

```
rep = record {back: int, items: array [item]}
```

Each element of *items* is initialized to a special *null* value, and *back* is initialized to 1. Enq and Deq are implemented as follows:

```
Enq = proc (q: queue, x: item)
  i: int := INC(q.back)   % Allocate a new slot.
  STORE (q.items[i], x)   % Fill it.
  end Enq

Deq = proc (q: queue) returns (item)
  while true do
    range: int := READ(q.back) − 1
    for i: int in 1 .. range do
      x: item := SWAP(q.items[i], null)
      if x ~= null then return(x) end
      end
    end
  end Deq
```

An Enq execution occurs in two distinct steps, which may be interleaved with steps of other concurrent operations: an array slot is reserved by atomically incrementing *back*, and the new item is stored in *items*.[3] Deq traverses the array in ascending order, starting at index 1. For each element, it atomically swaps *null* with the current contents. If the value returned is not equal to *null*,

[3] Like the FETCH-AND-ADD operation [30], INC returns the value of its argument from before the invocation, not the newly incremented value.

Deq returns that value, otherwise it tries the next slot. If the index reaches *q.back* − 1 without encountering a nonnull element, the operation is restarted. (Note that there is a small chance that a dequeuing process may starve if it is continually overtaken by other dequeuing processes. Any queue item, however, will eventually be dequeued as long as there are active dequeuers.) All atomic steps can be interleaved with steps of other operations. An interesting aspect of this implementation is that there is no mutual exclusion: no process can delay other processes by halting in a critical section. As an aside, we note that this implementation could be rendered more efficient by reclaiming slots from which items have been dequeued, reducing both the overall size of the rep of the queue and the cost of dequeuing an item. Such optimizations, however, would add nothing to our discussion of verification, so we ignore them in this paper.

The first difficulty arises when trying to define a rep invariant for this implementation. For sequential objects, the rep invariant must be satisfied at the start and finish of each abstract operation, but it may be violated temporarily while an operation is in progress. For concurrent objects, however, it no longer makes sense to view the object's representation as assuming meaningful values only between abstract operations. For example, our queue implementation permits operations to be in progress at every instant, thus the object may never be "between operations." When implementing a queue operation, one must be prepared to encounter a rep value that reflects the incomplete effects of concurrent operations, a problem that has no analog in the sequential domain. To assign a meaning to such transient values, the abstraction function must be defined continually, not just between abstract operations. As a consequence, the rep invariant must be preserved by each rep operation in the sequence implementing each abstract operation.

Another, more subtle difficulty arises when attempting to define an abstraction function. One natural approach is the following, proposed by Lamport [32]. A (continually defined) abstraction function **A** is chosen so that each abstract operation "takes effect" instantaneously at some step in its execution. In our queue example, when a process enqueues an item x, exactly one of the operations implementing the Enq would carry the rep from r to r', where $\mathbf{A}(r') = ins(\mathbf{A}(r), x)$. Surprisingly, perhaps, this technique fails to work for our queue implementation. To see why, we assume that such a function **A** exists, and we derive a contradiction. Consider the following scenario. Processes A and B invoke concurrent Enq operations, respectively enqueuing x and y. By incrementing the *back* counter, A reserves array position 1 and B reserves array position 2. B stores y in the array and returns. This computation is represented by the following history, where rep operations are indented and shown in upper-case.

```
Enq(x) A
Enq(y) B
  INC(q.back) A
  OK(1) A
  INC(q.back) B
  OK(2) B
  STORE(q.items[2], y) B
  OK( ) B
Ok( ) B
```

Let r be the rep value after this history. Because B's Enq operation has returned, $A(r)$ must reflect B's Enq. Because A's Enq operation is still in progress, $A(r)$ may or may not reflect A's Enq, depending on how A is defined. Thus, since no other operations have occurred, $A(r)$ must be one of $[y]$, $[y, x]$, or $[x, y]$, where the leftmost item is at the head of the queue.

We now derive a contradiction by showing that each of these values is contradicted by some future computation. First, assume $A(r)$ is $[x, y]$. If we now suspend A and allow a third process C to execute a Deq, C's Deq will return y, contradicting our assumption.

```
Deq( ) C
  READ(q.back) C
  OK(2) C
  SWAP(q.items[1], y) C
  OK(null) C
  SWAP(q.items[2], y) C
  OK(y) C
Ok(y) C
```

Second, assume $A(r)$ is $[y]$ or $[y, x]$. Allow A to complete its Enq, leaving a rep value r'. Now x must be in the queue, since its Enq is complete, and moreover it must follow y in the queue since, by hypothesis, A's enqueue appears to take effect after B's. It follows that $A(r')$ must be $[y, x]$. If C then executes a Deq, however, it will return x, a contradiction.

```
  STORE(q.items[1], x) A
  OK( ) A
Ok( ) A
Deq( ) C
  READ(q.back) C
  OK(2) C
  SWAP(q.items[1], y) C
  OK(x) C
Ok(x) C
```

The problem here is that the linearization order depends on a race condition: A's Enq will appear to occur before B's if A stores into location 1 before C reads from it, otherwise the order is reversed. Such nondeterminism is perfectly acceptable, however, because all resulting histories are linearizable. We circumvent this difficulty by redefining the abstraction function to map a rep value to a *set* of abstract values. This set represents the possible set of linearizations permitted by the current value of the rep. For objects that permit low levels of concurrrency, the value of the abstraction function might be a singleton set.

In conclusion, the rep invariant **I** must be continually satisfied and the abstraction function continually defined, not only between abstract operations, but also between rep operations implementing abstract operations. The abstraction function maps each rep value to a nonempty set of abstract values:

$$\mathbf{A}: \text{REP} \rightarrow 2^{\text{ABS}}$$

The nondeterminism inherent in a concurrent computation thus gives our notions of abstraction function and rep invariant a different flavor from their sequential counterparts.

4.3 Verification Method

In the next three sections we show how we use our new interpretation of representation invariant and abstraction function for proofs of correctness. We illustrate these ideas on the queue example presented in the previous section, as well as for an alternative implementation that uses critical sections.

4.3.1 *Linearized Values.* So far, linearizability is discussed in terms of histories. This characterization is useful for motivating the property, and for demonstrating properties such as locality, but it is awkward for verification. For linearizable histories, however, assertions about interleaved histories can be transformed into assertions about sets of sequential histories, and thus, sets of values. The transformed assertions can be stated and proved with the help of familiar axiomatic methods developed for sequential programs.

For a given history H, we call the value of an object at the end of a linearization of H a *linearized value.* Since a given history may have more than one linearization, an object may have more than one linearized value at the end of a history. We let *Lin(H)* denote the set of all linearized values of H. Informally, a history's linearized values represent the object's possible values from the point of view of an external observer. Figure 4 shows a queue history with its set of linearized values after each event. Initially, only the empty queue is associated with the empty history. After the invocation of Enq(x), there are two linearized values, since the enqueue may or may not have taken effect. After the invocation of Enq(y), there are five linearized values: either Enq may or may not have occurred, and if both have occurred, either ordering is possible. After the response to Enq(y), y is known to have been enqueued, and after the response to Enq(x), both x and y must have been enqueued, although their order remains ambiguous until x is dequeued.

4.3.2 *Proof Method.* To show correctness, the verification technique for sequential implementations is generalized as follows. Assume that the implementation of r is correct, hence H | REP is linearizable for all H in the implementation. Our verification technique focuses on showing the following property:

$$\text{For all } r \text{ in Lin(H | REP), } \mathbf{I}(r) \text{ holds and } \mathbf{A}(r) \subseteq \text{Lin(H | ABS)}$$

This condition implies that Lin(H | ABS) is nonempty, hence that H | ABS is linearizable. Note that the set inclusion is necessary in one direction only; there may be linearized abstract values that have no corresponding representation values. Such a situation arises when the representation "chooses" to linearize concurrent operations in one of several permissible ways.

4.3.3 *The Queue Example.* Returning to our queue example, our verification method is applied as follows. Let H | REP be a complete history for a queue representation, REP. If r is a linearized value for H | REP, define *items*(r) to be the set of non-null items in the array r.items. Let $<_r$ be the partial order such that $x <_r y$ if the STORE operation for x precedes the INC operation for y in H | REP. We can encode the partial order $<_r$ as auxiliary data. For a queue q, let $<_q$ denote the total order on its items, and *items*(q), the set of its items.

History	Linearized values
	{[]}
Enq(x) A	{[], [x]}
Enq(y) B	{[], [x], [y], [x,y], [y,x]}
Ok() B	{[y], [x,y], [y,x]}
Ok() A	{[x,y], [y,x]}
Deq() C	{[x], [y], [x,y], [y,x]}
Ok(x) C	{[y]}

Fig. 4. Linearized values.

The implementation has the following rep invariant:

$$\begin{aligned}\mathbf{I}(r) = {} & (r.\text{back} \geq 1)\\ & \wedge\ (\forall i.\ i \geq r.\text{back} \Rightarrow r.\text{items}[i] = \text{null})\\ & \wedge\ (\text{lbound}(r.\text{items}) = 1)\end{aligned}$$

where *lbound* is the lowest array index, and the following abstraction function:

$$\mathbf{A}(r) = \{q \mid \text{items}(r) = \text{items}(q) \wedge <_r \subseteq <_q\}$$

In other words, a queue representation value corresponds to the set of queues whose items are the items in the array, sorted in some order consistent with the precedence order of their Enq operations. Thus, our implementation allows for an item with a higher index to be removed from the array before an item with a lower index, but only if the items were enqueued concurrently.

Figure 5 shows a sequence of abstract operations of Figure 4 along with their implementing sequence of rep operations. Column two is the set of abstracted linearized rep values. Column three is the set of linearized abstract values. Our correctness criterion requires showing that each set in column two is a subset of the corresponding set in column three.

Appendix II outlines a complete formal proof of correctness (see also [45]). It relies on two key facts: (1) Enq enqueues an item x that is maximal with respect to $<_r$, and (2) Deq removes and returns an item x that is minimal with respect to $<_r$.

4.3.4 *Critical Sections.* So far our method for proving the correctness of an implementation assumes there exists a continually defined abstraction function. If the object's implementation includes critical sections, however, it may not always be possible to define such a function. Within the critical section, the rep invariant may be temporarily violated, leaving the abstraction function undefined. We show here how to overcome this difficulty relying on the standard trick of using (auxiliary) hidden data [37], thereby permitting us to reintroduce a continually defined abstraction function with the extended representation as its domain.

Both the problem and the solution are best illustrated by a simple example. Let us replace the atomic SWAP operation with a sequence of rotations executed within a critical section. Items are represented by 32-bit quantities, and the queue representation is expanded to associate a lock with each item:

```
rep = record{back: int, items: array[item],
locks: array[mutex]}
```

ROT(x, y) atomically rotates the 64-bit quantity by one bit. The Deq operation is implemented as follows:

```
Deq = proc(q: queue) returns (item)
  while true do
    range: int := READ(q.back)−1
    x: item := null
    for i: int in 1..range do
      LOCK(q.locks[i]) % start critical section
      for k: int in 1..32 do
        ROT(q.items[i], x)
        end
    UNLOCK(q.items[i]) % end critical section
    if x ~= null then return(x) end
    end
  end Deq
```

Although it is clear that this implementation is linearizable, its correctness cannot be proved directly using the method outlined so far. While the rotation is in progress, the abstraction function is undefined because necessary state information is encoded in the process's program counter and local variables, not in the representation itself. Thus, we introduce an auxiliary array of items to hold the value being shifted out of the queue, shown here as an additional field in the representation. Auxiliary data and statements are shown in italics. Statements enclosed in angle brackets are executed atomically.

```
rep = record{back: int
             items: array[item],
             aux: array[item],
             locks: array[mutex]
             }

Enq = proc(q: queue, x: item)
  i: int := INC(q.back)
  ⟨STORE(q.items[i], x)
  STORE(q.aux[i], x)⟩ % Make a redundant copy.
  end Enq

Deq = proc(q: queue) returns (item)
  while true do
    range: int := READ(q.back)−1
    x: item := null
    for i: int in 1..range do
      LOCK(q.locks[i]) % start critical section
      for k: int in 1..32 do
        ROT(q.items[i], x)
        end
    STORE(q.aux[i], null) % Update auxiliary array.
    UNLOCK(q.items[i]) % end critical section
    if x ~= null then return(x) end
    end
  end Deq
```

By embedding the representation object in an extended representation, we can give a continually defined abstraction function, one that agrees with the original abstraction function when the object is quiescent. We can use our proof method

History	A(Lin(H | REP))	Lin(H | ABS)
Enq(x) A	{[]}	{[], [x]}
INC(q.back) A	{[]}	{[], [x]}
OK(1) A	{[]}	{[], [x]}
STORE(q.items[1], x) A	{[], [x]}	{[], [x]}
OK() A	{[x]}	{[], [x]}
Enq(y) B	{[x]}	{[], [x], [y], [x,y], [y,x]}
INC(q.back) B	{[x]}	{[], [x], [y], [x,y], [y,x]}
OK(2) B	{[x]}	{[], [x], [y], [x,y], [y,x]}
STORE(q.items[2], y) B	{[x], [x,y]}	{[], [x], [y], [x,y], [y,x]}
OK() B	{[x,y]}	{[], [x], [y], [x,y], [y,x]}
Ok() B	{[x,y]}	{[y], [x,y], [y,x]}
Ok() A	{[x,y]}	{[x,y], [y,x]}
Deq() C	{[x,y]}	{[x,y], [y,x], [x], [y]}
READ(q.back) C	{[x,y]}	{[x,y], [y,x], [x], [y]}
OK(2) C	{[x,y]}	{[x,y], [y,x], [x], [y]}
SWAP(q.items[1], null) C	{[x,y], [y]}	{[x,y], [y,x], [x], [y]}
OK(x) C	{[y]}	{[x,y], [y,x], [x], [y]}
Ok(x) C	{[y]}	{[y]}

Fig. 5. A queue history.

to show the correctness of the extended representation, which then implies the correctness of the original.

The implementation has the following rep invariant:

$$\begin{aligned}\mathbf{I}(r) = {} & (r.\text{back} \geq 1) \\ & \wedge (\forall i.\ i \geq r.\text{back} \Rightarrow (r.\text{items}[i] = \text{null} \wedge r.\text{aux}[i] = \text{null})) \\ & \wedge (\forall i.\ (i < r.\text{back} \wedge r.\text{locks}[i] = \text{FREE}) \Rightarrow r.\text{items}[i] = r.\text{aux}[i]) \\ & \wedge (\text{lbound}(r.\text{items}) = 1 \wedge \text{lbound}(r.\text{aux}) = 1)\end{aligned}$$

The third conjunct is the most interesting since it states that the auxiliary array and the "real" array agree on all unlocked items.

Below, let $\mathbf{A}'$ be the extended abstraction function defined on the object r of the original rep type, and z, the auxiliary data. As before, we define $<_r$ to be the partial order on items in the r.items array, and similarly define $<_z$ to be the partial order on items in the r.aux array. The abstraction function is:

$$\begin{aligned}\mathbf{A}'(r, z) = \{q \mid {} & (\exists i.\ (i < r.\text{back} \wedge r.\text{locks}[i] \neq \text{FREE})) \\ & \Rightarrow (\text{items}(q) = \text{items}(z) \wedge <_z \subseteq <_q) \\ & \wedge (\forall i.\ (i < r.\text{back} \wedge r.\text{locks}[i] = \text{FREE})) \\ & \Rightarrow (\text{items}(q) = \text{items}(r) \wedge <_r \subseteq <_q)\}\end{aligned}$$

If a rotation is in progress the extended abstraction function simply uses the auxiliary value. When the object is quiescent, each lock is free, and $\mathbf{A}'$ agrees with the original $\mathbf{A}$.

5. REASONING ABOUT LINEARIZABLE OBJECTS

In the previous section we showed how to reason about the correctness of an implementation, given that linearizability is our correctness condition. In this section we show how we reason about properties of concurrent objects given just their (sequential) specifications and the assumption that they are implemented correctly, i.e., that they are linearizable.

5.1 Concurrent Registers

Here are axioms for Read and Write operations for all concurrent register objects, r:

$$\{\text{true}\}$$
$$\text{Read( )/Ok}(v)$$
$$\{r.\text{val} = r'.\text{val} = v\}$$
$$\{\text{true}\}$$
$$\text{Write}(v)\text{/Ok( )}$$
$$\{r'.\text{val} = v\}$$

These sequential axioms can be combined with our linearizability condition to prove assertions about the interleavings permitted by concurrent registers. Below, in a linearization H of a register history, let v_i denote the value of the register after the ith (complete) operation of H.

Every value read was written, but not overwritten.

THEOREM 3. *If r is a Read( )/Ok(v) operation in* H, *then there exists a Write(v)/Ok( ) operation w such that r does not precede w, and there is no other Write operation w' such that w precedes w' and w' precedes r.*

PROOF. Let r be the kth operation in a linearization of H, and let $i < k$ be the greatest index such that $v_i = v$. By construction, the ith operation in H is the Write(v) operation. If w' exists, then there exists j such that $i < j < k$ and $v_j \neq v$, a contradiction. □

Register values are persistent in the absence of Write operations.

THEOREM 4. *An* interval *in a history is a sequence of contiguous events. If I is an interval that does not overlap any Write operations, then all Read operations that lie within I return the same value.*

PROOF. Pick two Read operations e_i and e_j, $i < j$, that lie within the interval I. If $v_i \neq v_j$, then a Write operation must be linearized after e_i and before e_j, contradicting the assumption that no Writes overlap I. □

5.2 Concurrent Queues

The proofs of the following properties of concurrent queues use the following fact, which follows from Axioms E and D in Figure 3. For simplicity, we assume all values of items in a queue are unique.

LEMMA 5. *In any sequential queue history where x is enqueued before y, x is not dequeued after y.*

THEOREM 6. *If the Enq of x, Enq of y, Deq of x, and Deq of y are complete operations of* H *such that x's Enq precedes y's Enq, then y's Deq does not precede x's Deq (i.e., either x's Deq precedes y's, or they are concurrent).*

PROOF. Suppose not, i.e., y's Deq precedes x's Deq. Pick a linearization, and let q_i and q_j be queue values following the Deq operations of x and y respectively. From the assumption that $j < i$, $q_{j-1} = [y, \ldots, x, \ldots]$, which implies that y is enqueued before x, a contradiction. □

Gottlieb, Lubachevsky, and Rudolph [15] adopt the property proved in Theorem 6 as the (informal) correctness property for a linearizable queue implementation. The difficulty of reasoning informally about concurrent histories is illustrated by observing that Theorem 6 by itself is incomplete as a concurrent queue specification, since it does not prohibit implementations in which enqueued items spontaneously disappear from the queue, or new items spontaneously appear. Such behavior is easily ruled out by the following two theorems:

Items do not spontaneously vanish from the queue.

THEOREM 7. *If the Enq of x precedes the Enq of y, and if y has been dequeued, then either x has been dequeued or there is a pending Deq concurrent with the Deq of y.*

PROOF. Pick a linearization. Suppose x has not been dequeued. Let q_j be the value of the queue following the Deq of y. If y has been dequeued, but x has not, $q_{j-1} = [y, \ldots, x, \ldots]$, contradicting the assumption that the Enq of x precedes the Enq of y. □

Items do not spontaneously appear in the queue.

THEOREM 8. *If x has been dequeued, then it was enqueued, and the Deq operation does not precede the Enq.*

PROOF. Suppose not. Pick a linearization, and let q_i and q_j be the queue values after the Enq and Deq operations respectively. From our assumption, $j < i$. Then $q_{j-1} = [x, \ldots]$ and $q_i = [\ldots, x]$, implying by the uniqueness of the values of the items, that $i \leq j - 1 < j$, a contradiction. □

6. DISCUSSION

6.1 Related Work

The axiomatic approach to specifying sequential programs has its origins in Hoare's early work on verification [24]. Owicki and Gries extended Hoare's work to handle concurrent programs [37] by including axioms for general concurrent programming language constructs such as the parallel operator. Apt et al. [3] use an axiomatic approach for CSP [27]. Many researchers have also developed proof techniques for concurrent programs using conditional critical regions and monitors [7, 14, 28, 44]. We appeal to this past work when we perform syntax-directed reasoning about our implementations. In particular, we rely on standard techniques to deal with noninterference, using auxiliary data to encode both the program counters of other processes (e.g., the auxiliary array of Section 4.3.4) and history information (e.g., the $<_r$ partial order on items). All of this work, however, differs from ours by focusing on control structures. Data are either left completely uninterpreted or assumed to be of simple primitive types like booleans and integers. In contrast, our work on specifying and verifying concurrent objects focuses on data entirely, exploiting the semantics of the data type to increase the degree of concurrency. Our work builds upon, not replaces, older verification technology.

Related axiomatic work in abstract data types deals with proofs of correctness of their implementations [25], where, typically, first-order predicate logic pre- and post conditions are used for the specification of each operation of the type. Standish [43] and Nakajima [36] use a similar approach. The algebraic approach, which defines data types to be heterogeneous algebras [5], uses axioms to specify properties of programs and abstract data types, but the axioms are restricted to equations. Much work has been done on algebraic specifications for abstract data types [2, 8, 10, 17]. Any one of these approaches would be adequate for specifying the sequential behavior of a data type as required by our definition of when a sequential history is *legal.* In practice, we use Larch [19, 20]. Our contribution to the area of specifying abstract data types is that we can work with data in a concurrent, not just sequential, domain.

In short, whereas verification of concurrent programs focused on control, we focus on data; whereas past verification of abstract data types is applicable for sequential programs, ours is applicable for concurrent ones.

One notable exception is Lamport's work [32] in which he proposed a model and assertion language for specifying safety and liveness properties of concurrent objects. His approach is more general than ours, as it addresses liveness as well as safety properties, and nonlinearizable as well as linearizable behavior. Our approach, however, focuses exclusively on a subset of concurrent computations that we believe to be the most interesting and useful. In place of a specification language powerful enough to specify all conceivable concurrent behaviors, we re-interpret assertions about "well-behaved" concurrent computations as assertions about their equivalent sequential computations.

Moreover, Lamport's technique is based on a continually defined abstraction function (called a state function) that maps the representation to a single abstract value. This abstraction function defines the instant at which each operation appears to take effect: each primitive step of each operation either leaves the function's value unchanged, or it instantaneously causes the operation to take effect. This technique is not powerful enough to verify highly concurrent objects such as the queue implementation given in Section 4. Indeed, our linearizable queue example has since inspired Abadi and Lamport to extend Lamport's original technique to include not only history variables, but *prophecy* variables [1]. Prophecy variables are related to hidden variables called *possibilities* which we use in our proofs in the Appendices.

Our notion of linearizability generalizes and unifies similar notions found in specific examples in the literature. The use of concurrency control mechanisms such as monitors [26] or Ada tasks [9] is usually illustrated by simple implementations of linearizable objects such as bounded FIFO queues. These implementations permit very little concurrency, since operations execute one at a time. A more interesting example is due to Lamport [32], who verifies linearizability and liveness for a queue implementation that permits one enqueuing process to execute concurrently with one dequeuing process. There exists extensive literature on concurrent B-trees [4, 33, 42] and related search structures [6, 11, 13, 16, 29]. Although the correctness properties for these data structures are often stated in ad hoc terms, it is clear that they are meant to be linearizable. The algorithms cited above provide excellent additional examples of nontrivial techniques for implementing linearizable objects.

Misra [35] has proposed an axiomatic treatment of concurrent hardware registers in which the register's value is expressed as a function of time. Restricted to registers, our axiomatic treatment is equivalent to his in the sense that both characterize the full set of linearizable register histories. Theorems 3 and 4 capture two properties of Misra's registers. Misra's explicit use of time in axioms is appropriate for hardware, where reasoning in terms of the register's hypothetical value is useful as a guide to hardware designers. Our approach, however, is also appropriate for objects implemented in software, as we have found that reasoning directly in terms of partial orders generalizes more effectively to data types having a richer set of operations.

Gottlieb et al. [15] have investigated architectural support for implementing concurrent objects without critical sections, an approach illustrated by our linearizable implementation of a FIFO queue. They present a linearizable implementation of a concurrent queue (different from ours). The correctness condition asserted for their queue, however, is the property stated in Theorem 6, which by itself is incomplete as a concurrent queue specification since it does not prohibit implementations in which enqueued items spontaneously disappear from the queue, or new items spontaneously appear. As shown by Theorems 7 and 8, such anomalous behavior is easily ruled out by our queue axioms and the assumption of linearizability.

6.2 Final Remarks

Without linearizability, the meaning of an operation may depend on how it is interleaved with concurrent operations. Specifying such behavior would require a more complex specification language, as well as producing more complex specifications. Linearizability provides the illusion that each operation takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can still be given by pre- and post conditions.

The role of linearizability for concurrent objects is analogous to the role of linearizability for database theory: it facilitates certain kinds of formal (and informal) reasoning by transforming assertions about complex concurrent behavior into assertions about simpler sequential behavior. Like serializability, linearizability is a safety property; it states that certain interleavings cannot occur, but makes no guarantees about what must occur. Other techniques, such as temporal logic [32, 34, 39], must be used to reason about liveness properties such as fairness or priority.

An implementation of a concurrent object need not realize all interleavings permitted by linearizability, but all interleavings it does realize must be linearizable. The actual set of interleavings permitted by a particular implementation may be quite difficult to specify at the abstract level, being the result of engineering trade-offs at lower levels. As long as the object's client relies only on linearizability to reason about safety properties, the object's implementor is free to support any level of concurrency that appears to be cost-effective.

In conclusion, linearizability provides benefits for specifying, implementing, and verifying concurrent objects in multiprocessor systems. Rather than introducing complex new formalisms to reason directly about concurrent computations, we feel it is more effective to transform problems in the concurrent domain into simpler problems in the sequential domain.

I. GENERAL PROOFS OF CORRECTNESS

The proofs of the lemmas in this section are given elsewhere [45].

I.1 Possibilities and Linearized Values

For each linearized value, it is sometimes useful to keep track of which invocations were completed in the linearization that yielded that value, and what their responses were. A *possibility* for a history H is a triple $\langle v, \mathrm{P}, \mathrm{R}\rangle$, where v is a linearized value of H, P is the subset of pending invocations in H *not* completed when forming the linearization that yielded v, and R is the set of responses appended to H to form v. We let *Poss*(*H*) denote the set of possibilities of a history H. The relationship between the set of possibilities and set of linearized values for a given history H is the following: for each $\langle v, \mathrm{P}, \mathrm{R}\rangle$ $v \in \mathrm{Poss(H)}$, $v \in \mathrm{Lin(H)}$. For the example in Figure 4, the possibilities $\langle [\,], \{\mathrm{Enq}(x)\ \mathrm{A}\}, \varnothing\rangle$ and $\langle [x], \varnothing, \{\mathrm{Ok}(\,)\ \mathrm{A}\}\rangle$ are in $\mathrm{Poss}(\langle \mathrm{Enq}(x)\ \mathrm{A}\rangle)$. In the first case, the linearization is the empty history: the queue is empty, the pending Enq invocation was not completed, and no responses were appended. In the second case, the linearization is a single Enq operation: the queue holds x, no pending invocations were left incomplete, and A's Enq was completed normally. Similarly, $\langle [x, y], \varnothing, \{\mathrm{Ok}(\,)\ \mathrm{A}, \mathrm{Ok}(\,)\ \mathrm{B}\}\rangle$ and $\langle [y, x], \varnothing, \{\mathrm{Ok}(\,)\ \mathrm{A}, \mathrm{Ok}(\,)\ \mathrm{B}\}\rangle$ are two of the possibilities (among many others) in $\mathrm{Poss}(\langle \mathrm{Enq}(x)\ \mathrm{A}\rangle \cdot \langle \mathrm{Enq}(y)\ \mathrm{B}\rangle)$.

I.2 Four Generic Axioms

In order to carry out a formal proof of correctness for our queue example, it helps to appeal to the following four type-independent axioms. These axioms are used to derive a history's set of possibilities, and hence its set of linearized values.

Let x be the object whose operations appear in H. The following *closure axiom* states that if v is in Lin(H) and $\langle \mathrm{inv\ A}\rangle$ is a pending invocation in H that is not completed to form v, but could be completed with a response $\langle \mathrm{res\ A}\rangle$ to yield a legal value v' for x, then v' is also in Lin(H):

Axiom C:

$$\begin{aligned}&\langle v, \mathrm{P}, \mathrm{R}\rangle \in \mathrm{Poss(H)} \wedge \langle \mathrm{inv\ A}\rangle \in \mathrm{P} \wedge \{x = v\}\ \mathrm{inv/res}\ \{x = v'\}\\ &\quad \Rightarrow \langle v', \mathrm{P} - \{\mathrm{inv\ A}\}, \mathrm{R} \cup \{\mathrm{res\ A}\}\rangle \in \mathrm{Poss(H)}\end{aligned}$$

We write "$\{x = v\}$ inv/res $\{x = v'\}$" to indicate that the condition must be derivable from the sequential axioms for x.

The following *invocation axiom* states that any linearization of H is also a linearization of H · ⟨inv A⟩:

Axiom I:

$$\begin{aligned}&\langle v, \mathrm{P}, \mathrm{R}\rangle \in \mathrm{Poss}(\mathrm{H})\\ &\quad\Rightarrow \langle v, \mathrm{P} \cup \{\mathrm{inv\ A}\}, \mathrm{R}\rangle \in \mathrm{Poss}(\mathrm{H}\cdot\langle \mathrm{inv\ A}\rangle)\end{aligned}$$

The following *response axiom* states that any linearization of H in which the pending ⟨inv A⟩ is completed with ⟨res A⟩ is also a linearization of H · ⟨res A⟩:

Axiom R:

$$\begin{aligned}&\langle v, \mathrm{P}, \mathrm{R}\rangle \in \mathrm{Poss}(\mathrm{H}) \text{ and } \langle \mathrm{res\ A}\rangle \in \mathrm{R}\\ &\quad\Rightarrow \langle v, \mathrm{P}, \mathrm{R} - \{\mathrm{res\ A}\}\rangle \in \mathrm{Poss}(\mathrm{H}\cdot\langle \mathrm{res\ A}\rangle)\end{aligned}$$

The following *initialization axiom* states that the possibility for the initial value v_0 of an object corresponds to the empty history.

Axiom S:

$$\{\langle v_0, \varnothing, \varnothing\rangle\} = \mathrm{Poss}(\Lambda)$$

For each operation of a typed object, Axioms C, I, R, and S are instantiated to yield type-specific axioms.

For a given history H with m events, we use $\mathrm{Poss}_i(\mathrm{H})$ to denote the set of possibilities for the ith prefix of H, for $0 \le i \le m$. A *derivation* that shows that $\langle v, \mathrm{P}, \mathrm{R}\rangle \in \mathrm{Poss}_m(\mathrm{H})$ is a sequence of implications of the form:

$$\begin{aligned}&\langle v_0, \mathrm{P}_0, \mathrm{R}_0\rangle \in \mathrm{Poss}_0(\mathrm{H})\\ &\quad\Rightarrow \ldots\\ &\quad\Rightarrow \langle v_j, \mathrm{P}_j, \mathrm{R}_j\rangle \in \mathrm{Poss}_k(\mathrm{H})\\ &\quad\Rightarrow \ldots\\ &\quad\Rightarrow \langle v_n, \mathrm{P}_n, \mathrm{R}_n\rangle \in \mathrm{Poss}_m(\mathrm{H})\end{aligned}$$

where $v_n = v$, $\mathrm{P}_n = \mathrm{P}$, $\mathrm{R}_n = \mathrm{R}$, and each implication is justified by Axiom C, I, or R.

Intuitively, a derivation is like a history. Each implication in a derivation is like a step in a proof, and each such step is justified by an axiom.

The axioms C, I, R, and S are *sound*:

THEOREM 9. *If there exists a derivation showing that* $\langle v, \mathrm{P}, \mathrm{R}\rangle$ *is a possibility for* H, *then* v *is a linearized value for* H.

Axioms C, I, R, and S are *complete*.

THEOREM 10. *If* $v \in \mathit{Lin}(\mathrm{H})$, *then there exists a derivation that* $\langle v, \mathrm{P}, \mathrm{R}\rangle \in \mathit{Poss}(\mathrm{H})$.

II. PROOF OF CORRECTNESS FOR THE QUEUE

II.1 Two Lemmas About Concurrent Queues

In a derivation, an *Enq inference* for x is an instantiation of Axiom C of the form:

$$\begin{aligned}&\langle q_j, \mathrm{P}_j, \mathrm{R}_j\rangle \in \mathrm{Poss}_k\\ &\quad\Rightarrow \langle \mathrm{ins}(q_j, x), \mathrm{P}_j - \{\mathrm{Enq}(x)\ \mathrm{A}\}, \mathrm{R}_j \cup \{\mathrm{Ok}(\,)\ \mathrm{A}\}\rangle \in \mathrm{Poss}_k\end{aligned}$$

A *Deq inference* is defined analogously.

Two inferences *commute* in a derivation if their order can be reversed without invalidating the derivation. A derivation showing $\langle q, \mathrm{P}, \mathrm{R}\rangle \in \mathrm{Poss}_m$ is in *canonical form* if each Enq inference for an item in q occurs "as late as possible," i.e., it does not commute with the next inference in the derivation.

Lemma 11 implies that if x is in q, the event following the Enq inference for x is either the return event for x, or the return event for an item that follows x in q.

LEMMA 11. *If δ is a canonical derivation showing that $\langle q, \mathrm{P}, \mathrm{R}\rangle \in Poss_m$, and x is an item in q, then the inference following the Enq inference for x is either the Enq inference for the item following x in q, or an application of Axiom* R *for the matching response to Enq(x).*

Lemma 12 states that we can consider equivalence classes of queues rather than individual queues.

LEMMA 12. *If $\langle q, \mathrm{P}, \mathrm{R}\rangle \in Poss_m$, and q^* is a queue value constructed by rearranging the items of q in an order consistent with the partial precedence order of their Enq operations, then $\langle q^*, \mathrm{P}, \mathrm{R}\rangle \in Poss_m$.*

II.2 Main Proof

Figure 6 shows the Enq and Deq implementation annotated with assertions that are true before and after each abstract invocation and response and each rep operation. To avoid distraction, we assume queue values are unique. It is convenient to keep as implicit auxiliary data the partial order, $<_r$, on items in the array, defined in Section 4.3.3. The set of possibilities, Poss, referred to in the annotations can also be encoded as auxiliary data in terms of the sets, P (pending invocations) and R (possible responses), which are components of a possibility.

If I is a set of items partially ordered by $<$, define:

$$(\mathrm{I}, <) = \{q \mid \mathrm{I} = \mathrm{items}(q) \text{ and } < \subseteq <_q\}$$

and

$$[(\mathrm{I}, <), \mathrm{P}, \mathrm{R}] = \{\langle q, \mathrm{P}, \mathrm{R}\rangle \mid q \in (\mathrm{I}, <)\}.$$

The partially ordered set of queue items (I, $<$), captures the nonquiescent abstract state of the queue, i.e., the possible values of the queue while there are concurrent Enq and Deq operations or pending invocations. Notice that we can rewrite the abstraction function as $\mathrm{A}(r) = (\mathrm{items}(r), <_r)$. The set $[(\mathrm{I}, <), \mathrm{P}, \mathrm{R}]$ identifies each of the possible sets of queue values with a set of pending

```
{$\exists\langle q, P, R\rangle \in$ Poss}
Enq = proc (q: queue, x: item)
{$\exists\langle q', P', R'\rangle \in$ Poss' . $q' = q \wedge P' \cup \{Enq(x)\ A\} \wedge R' = R$}
  {$\exists\langle q, P, R\rangle \in$ Poss . $\langle Enq(x)\ A\rangle \in P$}
  i : int := INC(q.back)
  {Poss' = Poss}
  {$\exists\langle q, P, R\rangle \in$ Poss . $\langle Enq(x)\ A\rangle \in P$}
  STORE(q.items[i], x)
  {$\exists\langle q', P', R'\rangle \in$ Poss' .$P' = P - \{Enq(x)\ A\} \wedge R' = R \cup \{Ok(\,)\ A\}$
      $\wedge$ index($q$.items', $x$) $= i \wedge x \in$ max(items($q'$)) $\wedge$ $q$.back $\leq$ $q$.back'}
  {$\exists\langle q, P, R\rangle \in$ Poss . $\langle Ok(\,)\ A\rangle \in R$}
  end Enq
  {$\exists\langle q', P', R'\rangle \in$ Poss'.$q' = q \wedge P' = P \wedge R' = R - \{Ok(\,)\ A\}$}

{$\exists\langle q, P, R\rangle \in$ Poss}
Deq = Proc (q: queue) returns (item)
{$\exists\langle q', P', R'\rangle \in$ Poss' . $q' = q \wedge P' = P \cup \{Deq(\,)\ A\} \wedge R' = R$}
  {$\exists\langle q, P, R\rangle \in$ Poss . $\langle Deq(\,)\ A\rangle \in P$}
  while true do
    range: int := READ(q.back) - 1
    {Poss' = Poss}
    for i: int in 1 .. range do
      {$\exists\langle q, P, R\rangle \in$ Poss . $\langle Deq(\,)\ A\rangle \in P$}
        x: item := SWAP(q.items[i], null)
        {$\exists\langle q', P', R'\rangle \in$ Poss' .$P' = P - \{Deq(\,)\ A\} \wedge R' = R \cup \{Ok(x)\ A\} \wedge$
          $(x = null \vee x \in$ min(items($q'$)))}
        if x ~= null then return(x) end
        end
      end
  end
  {$\exists\langle q, P, R\rangle \in$ Poss . $\langle Ok(x)\ A\rangle \in R$}
  end Deq
  {$\exists\langle q', P', R'\rangle \in$ Poss' . $q' = q \wedge P' = P \wedge R' = R - \{Ok(x)\ A\}$}
```

Fig. 6. Annotated queue implementation.

invocations and a set of possible responses, thereby forming a set of (queue) possibilities. The following two lemmas make use of Lemma 12, stated in the previous section.

LEMMA 13. *If x is a maximal element with respect to* $<$, $x \notin$ I, $\langle Enq(x)\ A\rangle \notin$ P, $\langle Ok(\,)\ A\rangle \in$ R, *and* $[(I, <), P \cup \{Enq(x)\ A\}, R - \{Ok(\,)\ A\}] \subseteq$ *Poss, then* $[(I \cup \{x\}, <), P, R] \subseteq$ *Poss.*

LEMMA 14. *If* $\langle Deq(\,)\ A\rangle \notin$ P, $\langle Ok(x)\ A\rangle \in$ R, *and* $[(I, <), P \cup \{Deq(\,)\ A\}, R - \{Ok(x)\ A\}] \subseteq$ *Poss, then for all x such that x is a minimal element of* I, $[(I - \{x\}, <), P, R] \subseteq$ *Poss.*

Lemma 13 will allow us to show that the set of linearized queue values does not change over a STORE operation and similarly, Lemma 14, for a SWAP operation, by using $<_r$ for $<$ and by recalling that for each $\langle v, P, R\rangle \in$ Poss, v is a linearized value. We use the next two lemmas to satisfy the conditions of the previous two lemmas.

LEMMA 15. *Enq enqueues an item x that is maximal with respect to* $<_r$.

LEMMA 16. *Deq removes and returns an item x that is minimal with respect to* $<_r$.

Here is a proof of correctness.

THEOREM 17. *The queue implementation is correct.*

PROOF. Assuming every rep history is linearizable, we need to show that every queue history, $H \mid q$, is linearizable. It suffices to show that the "subset" property, $\bigcup_{r \in \mathrm{Lin}(H \mid r)} \mathbf{A}(r) \subseteq \mathrm{Lin}(H \mid q)$, remains invariant over abstract invocation and responses and over complete rep operations. Thus, it can be conjoined to the pre- and post conditions of Figure 6 as justified by the Owicki–Gries proof method [38]. Axioms I and R give us the result for abstract invocation and response events. INC and READ leave the abstraction function the same. Thus, we are left with two cases, STORE and SWAP. By Lemma 15 we know that STORE adds a maximal item and thus, we can apply Lemma 13 to show that the subset property is preserved. Similarly, by Lemma 16 we know that SWAP removes a minimal item and thus, we can apply Lemma 14 to show that the subset property is preserved. □

ACKNOWLEDGMENTS

The authors thank Jim Horning, Leslie Lamport, Larry Rudolph, and William Weihl for lively verbal and electronic discussions about our notions of linearizability and correctness. We also thank James Aspnes, Stewart Clamen, David Detlefs, Richard Lerner, and Mark Maimone for their comments on earlier versions of this paper. Finally, we would like to thank Jim Gray and the anonymous referees for their comments and suggestions.

REFERENCES

1. ABADI, M., AND LAMPORT, L. The existence of refinement mappings. Tech. Rep. 29, DEC Systems Research Center, Aug. 1988.
2. GOGUEN, J. A., THATCHER, J. W., WAGNER, E. G., AND WRIGHT, J. B. Abstract data types as initial algebras and correctness of data representations. In *Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structures* (May 1975). ACM, New York, 1975, 89–93.
3. APT, K. R., FRANCEZ, N., AND DEROEVER, W. P. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst. 2*, 3 (July 1980), 359–385.
4. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. *Acta Inf. 1*, 1 (1977), 1–21.
5. BIRKHOFF, G., AND LIPSON, J. D. Heterogeneous algebras. *J. Comb. Theor. 8* (1970), 115–133.
6. BISWAS, J., AND BROWNE, J. C. Simultaneous update of priority structures. In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, Ill., 1987). 124–131.
7. BROOKES, S. D. An axiomatic treatment of a parallel language. In *Proceedings of Conference on Logics of Programs. Lecture Notes in Computer Science. Vol. 193.* Springer-Verlag, Berlin, 1985.

8. BURSTALL, R. M., AND GOGUEN, J. A. Putting theories together to make specifications. In *Fifth International Joint Conference on Artificial Intelligence* (Cambridge, Mass., Aug. 1977). 1045–1058. Invited paper.
9. DEPARTMENT OF DEFENSE. *Reference Manual for the ADA Programming Language.* ANSI/ MIL-STD-1815A-1983, 1983.
10. EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.
11. ELLIS, C. S. Concurrent search and insertion in 2-3 trees. *Acta Inf. 14* (1980), 63–86.
12. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notion of consistency and predicate locks in a database system. *Commun. ACM 19,* 11 (Nov. 1976), 624–633.
13. FORD, R., AND CALHOUN, J. Concurrency control mechanisms and the serializability of concurrent tree algorithms. In *3rd ACM Symposium on Principles of Database Systems* (1984). ACM, New York, 1984, 51–60.
14. GERTH, R., AND DEROEVER, W. P. Proving monitors revisited: A first step towards verifying object oriented systems. *Fundamental Inf. 9* (1986), 371–400.
15. GOTTLIEB, A., LUBACHEVSKY, B. D., AND RUDOLPH, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst. 5,* 2 (April 1983), 164–189.
16. GUIBAS, L., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In *19th ACM Symposium on Foundations of Computer Science* (Providence, R.I., 1978). ACM, New York, 1978, 8–21.
17. GUTTAG, J. V. The specification and application to programming of abstract data types. Ph.D. thesis, Univ. of Toronto, Toronto, Sept. 1975.
18. GUTTAG, J. V., HOROWITZ, E., AND MUSSER, D. R. Abstract data types and software validation. *Commun. ACM 21,* 12 (Dec. 1978), 1048–1064.
19. GUTTAG, J. V., HORNING, J. J., AND WING, J. M. Larch in five easy pieces. Tech. Rep. 5, DEC Systems Research Center, July 1985.
20. GUTTAG, J. V., HORNING, J. J., AND WING, J. M. The Larch family of specification languages. *IEEE Softw. 2,* 5 (Sept. 1985), 24–36.
21. HERLIHY, M., AND WING, J. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages* (Jan. 1987). ACM, New York, 1987, 13–26.
22. HERLIHY, M. P. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst. 12,* 2 (June 1987), 170–194.
23. HERLIHY, M. P. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (*PODC*) (Toronto, Ont., Aug. 1988). ACM, New York, 1988, 276–290.
24. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12,* 10 (Oct. 1969), 576–583.
25. HOARE, C. A. R. Proof of correctness of data representations. *Acta Inf. 1,* 1 (1972), 271–281.
26. HOARE, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17,* 10 (Oct. 1974), 549–557.
27. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21,* 8 (Aug. 1978), 666–677.
28. HOWARD, J. H. Proving monitors. *Commun. ACM 19,* 5 (May 1976), 273–279.
29. JONES, C. B. *Software Development: A Rigorous Approach.* Prentice-Hall, Englewood Cliffs, N.J., 1980.
30. KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Aug. 1986). ACM, New York, 1986.
31. LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28,* 9 (Sept. 1979), 690–691.
32. LAMPORT, L. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5,* 2 (April 1983), 190–222.
33. LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst. 6,* 4 (Dec. 1981), 650–670.
34. MANNA, Z., AND PNUELI, A. Verification of concurrent programs, Part I: The temporal framework. Tech. Rep. STAN-CS-81-836, Dept. of Computer Science, Stanford Univ., June 1981.

35. MISRA, J. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst. 8*, 1 (Jan. 1986), 142–153.
36. NAKAJIMA, R., HONDA, M., AND NAKAHARA, H. Hierarchical program specification and verification—A many-sorted logical approach. *Acta Inf. 14* (1980), 135–155.
37. OWICKI, S., AND GRIES, D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM 19*, 5 (May 1976), 279–285.
38. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Acta Inf. 6*, 4 (1976), 319–340.
39. OWICKI, S., AND LAMPORT, L. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 455–495.
40. PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
41. REED, D. P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 3–23.
42. SAGIV, Y. Concurrent operations on B-trees with overtaking. In *Symposium on Principles of Database Systems* (Waterloo, Ont., Jan. 1985). ACM, New York, 1985, 28–37.
43. STANDISH, T. A. Data structures: An axiomatic approach. Rep. 2639, Bolt, Beranek, and Newman, Cambridge, Mass., Aug. 1973.
44. STIRLING, C. A generalization of Owicki–Gries–Hoare logic for a concurrent while language. Tech. Rep., Edinburgh Univ., March 1987.
45. HERLIHY, M. P., AND WING, J. M. Axioms for concurrent objects. Tech. Rep. CMU-CS-86-154, Computer Science Dept., Carnegie Mellon Univ., 1986.
46. WEIHL, W. E. Local atomicity properties: Modular concurrent control for abstract data types. *ACM Trans. Program. Lang. Syst. 11*, 2 (April 1989), 249–283.

Received January 1988, revised November 1988 and July 1989, accepted October 1989