Linearizability: A Correctness Condition for Concurrent Objects

Maurice P. Herlihy and Jeannette M. Wing

30 March 1988
CMU-CS-88-120

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

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 verifying implementations of linearizable objects, and shows how to reason about concurrent objects using their (sequential) axiomatic specifications.

A preliminary version of this paper appeared in the proceedings of the 14th ACM Symposium on Principles of Programming Languages, January 1987[6].

This research was sponsored by IBM and the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864 (Amendment 20), under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB. Additional support for J. M. Wing was provided in part by the National Science Foundation under grant CCR-8620027.

1. Introduction

1.1. Overview

Informally, a concurrent system consists of a collection of sequential processes that communicate through shared typed objects. This model is appropriate for multiprocessor systems in which processors communicate through reliable, high-bandwidth shared memory. Whereas "memory" suggests registers with read and write operations, we use the term *concurrent object* to suggest a richer semantics. 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. We can give an *axiomatic specification* for a typed object to define the meaning of its operations when they are invoked one at a time by a single process. 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 3, 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 known techniques from the sequential domain. Unlike alternative correctness conditions such as sequential consistency [11] or serializability [17], 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 *non-blocking* property: processes invoking totally-defined operations are never forced to wait. Non-blocking 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:

- We reason about the correctness of linearizable object implementations using new techniques that generalize the notions of representation invariant and abstraction function [3, 9] to the concurrent domain.
- 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 our specification techniques; Section 3 defines and discusses linearizability. Section 4 presents techniques for reasoning about implementations of linearizable objects, and Section 5 illustrates these techniques by verifying a novel implementation of a highly concurrent queue. Section 6 presents examples of how to use linearizability to reason about concurrent registers and queues. Section 7 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 non-concurrent operations should be 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 review some informal 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 1-1 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 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 1-1.a) 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 [14]. Our approach can be thought of as generalizing Misra's approach to objects with richer sets of operations. For example, H_5 in Figure 1-2a is acceptable, but H_6 is not (examples are taken from [14]). 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 two sections, we formalize the intuition presented here by defining the notion of linearizability to encompass those histories we have argued are intuitively acceptable.

```
       E(x) A                                      D(y) A         E(x) A
     |-----------|                               |----------| . |---------- ....

        E(y) B          D(x) B
       |-------|      |----------|
```

a. H_1 (acceptable).

```
     E(x) A                D(y) A
     |--------|            |----------|

                 E(y) B
                |----------|
```

b. H_2 (not acceptable).

```
     E(x) A
     |----------. ...

         D(x) B
        |----------|
```

c. H_3 (acceptable).

```
     E(x) A                   D(y) A
     |--------|               |----------|

          E(y) B                   D(y) C
         |----------|             |----------|
```

d. H_4 (not acceptable).

Figure 1-1: FIFO Queue Histories

2. System Model and Specification Technique

2.1. Histories

An execution of a concurrent system is modeled by a *history*, which is a finite sequence of operation *invocation* and *response events*. An operation invocation is written as ⟨*x op(args*) A*⟩, 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 ⟨*x term(res*) A*⟩, 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 longest subhistory of H consisting only of invocations and matching responses.

A history H is *sequential* if:

```
    W(0) A                 R(1) A          W(0) C
|-----------|          |-------------|  |------------|

                              W(1) B                            R(0) B
              |-------------------------------------|   |------------|
```

a. H_5 (acceptable).

```
    W(0) A                 R(1) A          W(0) C
|-----------|          |-------------|  |------------|

                              W(1) B                            R(1) B
              |-------------------------------------|   |------------|
```

b. H_6 (not acceptable).

Figure 1-2: Register Histories

1. The first event of H is an invocation.
2. Each invocation, except possibly the last, is immediately followed by a matching response.
3. Each response, except possibly the last, is immediately followed by an invocation.

A *process subhistory*, H | P (H at P), of a history H is the subsequence of 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 1-1 is the following well-formed history for a FIFO queue q.

```
q Enq(x) A                                              (History H1)
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.

2.2. Specifications

A sequential history for an object can be summarized by the object's value at the end of the history. We use standard axiomatic specifications to reason about object values, and hence, indirectly about sequential histories. A *specification* is a set of axioms of the form:

$$\{P\}$$
$$\text{op(args*) / term(res*)}$$
$$\{Q\}$$

where P is a pre-condition on the object's value and the argument values that must be met before an invocation, and Q is a post-condition on the object's value and the result values that is guaranteed to hold upon return for the given termination condition. Identifiers in args* and res* denote values of arguments and results. A sequential history H is *legal* if for all object subhistories, H | x, of H, each operation in H | x satisfies its axiomatic specification.

The axioms presented in this paper are essentially Larch interface specifications [4, 5] for operations of abstract data types. For example, axioms for the Enq and Deq operations for FIFO queues are shown in Figure 2-1. The queue's value before the operation is denoted by q and the value after the operation by q'. The post-condition for Enq states that upon 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, but applying it to an emptv queue returns an exception and leaves the queue unchanged. Notice that the specification for Deq is *partial:* Deq is undefined for the empty queue.

The assertion language for the pre- and post-conditions is based on the Larch Shared Specification Language. It is akin to algebraic specification languages and is used to describe the set of values of a typed objec' Figure 2-2 shows the Larch *trait* that defines queue values. The set of operators and their signatures following **Introduces** defines a vocabulary with which to compose terms that denote values. For example, *emp* and *ins(emp, 5)* denote two different queue values. The set of equations following the **constrains** 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 that allows one to prove properties of all queue values. We use the vocabulary of traits to write the assertions in the pre- and post-conditions of a type's operations; we use the meaning of equality to reason about its values. Hence, the meaning of *ins* and "=" in Axiom E's post-condition is given by the trait QVals.

3. Linearizability

This section defines the notion of linearizability, proves that it is a *local* and *non-blocking* property, and discusses the differences between it and other correctness conditions.

3.1. Definition

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

$e_0 <_H e_1$ if $res(e_0)$ precedes $inv(e_1)$ in H.

(Where appropriate, the subscript is 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

Axiom E:

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

Axiom D:

$$\{\neg\ \text{isEmp}(q)\}$$
$$\text{Deq}()\ /\ \text{Ok}(e)$$
$$\{q' = \text{rest}(q) \wedge e = \text{first}(q)\}$$

Figure 2-1: Axioms for Queue Operations

```
QVals: trait
  introduces
    emp: → Q
    ins: Q, E → Q
    first: Q → E
    rest: Q → Q
    isEmp: Q → Bool
 constrains Q so that
    Q generated by [ emp, ins ]
    for all q: Q, e: E
      first(ins(emp, 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
```

Figure 2-2: Trait for Queue Values

a total order.

A history H is *linearizable* if 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: $<_{H'} \subseteq <_S$.

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. Non-determinism 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.

3.2. Queue Examples Revisited

Let "·" denote concatenation of events. The history H_1 shown in Figure 1-1 is linearizable, because $H_1 \cdot \langle q\ \text{Ok}()\ A\rangle$ 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 H_2)
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 H_3)
q Deq() B
q Ok(x) B

H_3 can be extended to $H_3' = H_3 \cdot \langle$q 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 H_4)
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, hence H_4 is not equivalent to any sequential FIFO queue history.

3.3. Locality

Linearizability is a *local* property.

Theorem 1: H is linearizable if and only if H | x is linearizable at each object x.

Proof: The "only if" part is obvious.

From the assumption that each object's history is linearizable, there exists for each object x an induced total order $<_x$ on its own operations, and by the well-formedness criteria for histories, each process P induces a total order $<_P$ on its operations. We claim that the transitive closure of the union of all $<_x$ and $<_P$ is a partial order, $<$, and hence can be extended to a total order, $<$. Notice that each $<_x$ and $<_P$ is compatible with $<$.

Suppose < is not a partial order. Then we can construct a cycle of operations $e_1 \cdot \ldots \cdot e_n$, where $e_1 = e_n$, such that $e_1 < e_2 < \ldots < e_n$, where e_{i-1} and e_i, $1 < i \leq n$, are related by some $<_x$ or $<_P$. The contradiction is immediate if no pair is related by a $<_P$, because then all relations are induced by the same $<_x$, which is assumed to be a total order. Otherwise, we can assume without loss of generality that $e_1 <_P e_2$ for some process P. Because processes are sequential, the response of e_1 precedes the invocation of e_2, and because all relations are consistent with <, the invocation of e_2 precedes the response of e_n, which is identical to the response of e_1. Hence the response of e_1 precedes itself, a contradiction.

Henceforth, we need only consider histories involving single objects.

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 non-local 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.

3.4. Blocking versus Non-Blocking

Linearizability is a *non-blocking* property:

Theorem 2: Let inv be an invocation of an operation whose sequential specification is total, i.e., it has a response defined for every value. If ⟨x inv P⟩ is a pending invocation in a linearizable history H, then there exists a response ⟨x res P⟩ such that H · ⟨x res P⟩ is linearizable.

Proof: Let S be any linearization of H. If S includes a response to ⟨x inv P⟩, we are done. Otherwise, because x is total, there exists a response ⟨x res P⟩ 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 because [x inv/res P] precedes no other operation, it 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 it is not inherent to the correctness property itself. (Techniques for constructing non-blocking implementations of linearizable objects are discussed elsewhere [8].) 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 non-blocking property.

The non-blocking 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.

3.5. Comparison to Other Correctness Conditions

Lamport's notion of *sequential consistency* [11] requires that a history be equivalent to a 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 is not.

Much work on databases and distributed systems uses *serializability* [17] 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. 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 [1], but not by others, such as multi-version timestamp schemes [18], or schemes that provide high levels of availability in the presence of network partitions [7].

To compare serializability and linearizability, we must draw a correspondence between the elements of the two models. Processes do *not* correspond to transactions. Processes may appear to be interleaved, while transactions may not. Moreover, processes may run forever, while transactions must eventually terminate if they are to accomplish any useful work. Instead, a process corresponds roughly to a sequence of transactions, and a transaction corresponds to a primitive operation on a single object. Linearizability can thus be viewed as strict serializability where transactions are restricted to consist of a single operation applied to a single object. We will see that this single-operation restriction has important consequences, both formal and practical.

[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.

One important formal difference between linearizability and serializability is that neither serializability nor strict serializability is a local property. In the history H_8 shown above, both H_8 | p and H_8 | 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 dequeue 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, one object should not use two-phase locking while another uses multi-version timestamping.

Another important formal difference is that serializability places much 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 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 it is impossible to complete both pending invocations 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 is not a blocking property: 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 applications such as databases that must preserve complex application-specific invariants spanning multiple objects. A property preserved by transactions executing in isolation will be preserved by transactions executing concurrently. Linearizability, by contrast, is intended for applications such as multiprocessor operating systems in which concurrency is the primary interest, and the ease of preserving multi-object invariants is less important.

3.6. Derived Definitions: Linearized Values and Possibilities

So far, linearizability is discussed in terms of histories. This historic (!) 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 linearized values of H.

A linearized value for an object summarizes the set of possible histories that can yield that value. A *possibility* for a history H is a triple ⟨v, P, R⟩, where v is a linearized value of H, P is the subset of pending invocations in H that are *not* completed to form 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 ⟨v, P, R⟩ ∈ Poss(H), v ∈ Lin(H).

Informally, a history's linearized values represent the object's possible values from the point of view of an external observer. Figure 3-1 shows a queue history with its set of linearized values after each event.[2]

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]}

Figure 3-1: Linearized Values

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. The possibilities, ⟨[], {Enq(x) A}, ∅⟩ and ⟨[x], ∅, {Ok() A}⟩ are in Poss(⟨Enq(x) A⟩); two possibilities (among many others), ⟨[x, y], ∅, {Ok() A, Ok() B}⟩ and ⟨[y, x], ∅, {Ok() A, Ok() B}⟩ are in Poss(⟨Enq(x) A⟩ · ⟨Enq(y) B⟩).

4. Verifying Linearizability

In this section, we motivate and describe our method for verifying implementations of linearizable objects.

4.1. Representation Invariant and Abstraction Function

We begin by reviewing how to verify the correctness of sequential objects [3, 9]. In the sequential domain, an implementation consists of an *abstract type* A, the type being implemented, and a *representation* (or *rep*) type R, the type used to implement A. The subset of R values that are *legal* representations is characterized by a predicate called the *rep invariant*, *I*: R → bool. The meaning of a legal rep is given by an *abstraction function*, *A*: R → A, defined for rep 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 pre-condition and post-condition 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

[2]When convenient, we use [] for emp and [x,y] for ins(ins(emp, x),y), etc.

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 *A*(r) to *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*, Deq returns that value, otherwise it tries the next slot. If the index reaches *q.back-1* without encountering a non-null element, the operation is restarted. (Note there is a small chance that a dequeuing process may starve if it is continually overtaken by other dequeuing processes.) 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 is ever forced to wait for another. 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

[3]Like the FETCH-AND-ADD operation [10], INC returns the value of its argument from before the invocation, not the newly incremented value.

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 [12]. 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 *A(r') = ins(A(r), x)*. Unfortunately, this technique fails to work for our queue implementation. To see why, we assume that one can define such a function *A*, 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 non-determinism 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 concurrency, 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 non-empty set of abstract values:

$$\mathcal{A}: R \rightarrow 2^{A}$$

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

4.2. Verification Method

In the next three sections, we define our notion of correctness and present our proof method, give a rep invariant and abstraction function for our FIFO queue example, and give a set of "generic" axioms that can be instantiated for any given type in order to carry out formal proofs of correctness.

4.2.1. Correctness and Proof Method

An *implementation* is a set of histories in which events of two objects, a *representation* object REP of type R and an *abstract* object ABS of type A, 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. Informally, an abstract operation is implemented by the sequence of rep operations that occur within it.

An implementation is *correct* if for every history H in the implementation, H | ABS is linearizable.

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:

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

This condition implies that Lin(H | ABS) is non-empty, 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.

4.2.2. 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. Finally, we extend the trait of Figure 2-2 by defining the total order, $<_q$, and the operator, *items*, such that:

$$\text{first}(q) <_q^{'} \text{first}(\text{rest}(q))$$
$$\text{items}(\text{emp}) = \{\}$$
$$\text{items}(\text{ins}(q, e)) = \{e\} \cup \text{items}(q)$$

The implementation has the following rep invariant:

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

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

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

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 4-1 shows a sequence of abstract operations of Figure 3-1 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.

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]}
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], [y], [x,y]}	{[], [x], [y], [x,y], [y,x]}
Ok() B	{[y], [x,y]}	{[], [x], [y], [x,y], [y,x]}
Ok() B	{[y], [x,y]}	{[y], [x,y], [y,x]}
OK() A	{[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]}

Figure 4-1: A Queue History

4.2.3. 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 ⟨inv A⟩ is a pending invocation in H that is not completed to form v, but could be completed with a response ⟨res A⟩ to yield a legal value v' for x, then v' is also in Lin(H):

Axiom C:

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

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:

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

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:

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

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

$\langle v_0, P_0, R_0\rangle \in \mathrm{Poss}_0(H)$
$\Rightarrow \ldots$
$\Rightarrow \langle v_j, P_j, R_j\rangle \in \mathrm{Poss}_k(H)$
$\Rightarrow \ldots$
$\Rightarrow \langle v_n, P_n, R_n\rangle \in \mathrm{Poss}_m(H).$

where $v_n = v$, $P_n = P$, $R_n = R$, 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.

We first show that the axioms C, I, R, and S are *sound*:

Theorem 3: If there exists a derivation showing that ⟨v, P, R⟩ is a possibility for H, then v is a linearized value for H.

Proof: Given a derivation showing that $\langle v, P, R\rangle$ is a possibility for H, we claim that the order in which Axiom C is applied induces a valid linearization ordering on H, and hence that v is in Lin(H).

Let $\langle \text{inv } A\rangle$ be the ith event of H, and let the matching response $\langle \text{res } A\rangle$ be the jth event. Since the operation inv/res is complete, the derivation must include an application of Axiom C to infer:

$$\langle v', P, R\rangle \in \text{Poss}_k(H) \Rightarrow \langle v'', P - \{\text{inv } A\}, R \cup \{\text{res } A\}\rangle \in \text{Poss}_k(H).$$

First, we note that $i \le k$, since the only way to infer anything about $\text{Poss}_i(H)$ from $\text{Poss}_{i-1}(H)$ is to apply Axiom I as follows:

$$\langle u, P', R'\rangle \in \text{Poss}_{i-1}(H) \Rightarrow \langle u, P' \cup \{\text{inv } A\}, R'\rangle \in \text{Poss}_i(H)$$

Next, we note that $k < j$, since the only way to infer anything about $\text{Poss}_j(H)$ from $\text{Poss}_{j-1}(H)$ is by applying Axiom R:

$$\langle w, P'', R''\rangle \in \text{Poss}_{j-1}(H) \text{ and } \langle \text{res } A\rangle \in R'' \Rightarrow \langle w, P'', R'' - \{\text{res } A\}\rangle \in \text{Poss}_j(H).$$

Between these two steps, the only way to remove $\langle \text{inv } A\rangle$ from the set of pending invocations is by applying Axiom C as shown above. It follows that if one operation precedes another, then the first operation's application of Axiom C must precede the second's, hence the order in which Axiom C is applied to operations is compatible with the natural precedence order, and thus induces a linearization order.

We next show that Axioms C, I, R, and S are *complete*.

Theorem 4: If $v \in \text{Lin}(H)$, then there exists a derivation that $\langle v, P, R\rangle \in \text{Poss}(H)$.

Proof: By induction on the length of H. The base case is immediate, so we assume the result for histories of length n. Let H be a history of length n+1.

If $H = H' \cdot \langle \text{res } A\rangle$, then $\text{Lin}(H) \subseteq \text{Lin}(H')$. From the induction hypothesis, there exists a derivation that $\langle v, P, R\rangle \in \text{Poss}(H')$, and an application of Axiom R yields a derivation that $\langle v, P, R - \{\text{res } A\}\rangle \in \text{Poss}(H)$.

Suppose $H = H' \cdot \langle \text{inv } A\rangle$. Let L be a linearization of H that has v as its final value. If $\langle \text{inv } A\rangle$ does not appear in L, then, from the induction hypothesis, we can derive that $\langle v, P, R\rangle \in \text{Poss}(H')$, and a further application of Axiom I yields that $\langle v, P \cup \{\text{inv } A\}, R\rangle \in \text{Poss}(H)$. Otherwise, suppose L has k operations, where $\langle \text{inv } A\rangle$ is the ith invocation. Let L' be the prefix of L that precedes that invocation, and let v' be the value at the end of L'. Because any operation linearized after $\langle \text{inv } A\rangle$ in L must have an invocation in H but not a response, L' is a linearization of H', and, by the induction hypothesis, we can derive that $\langle v', P, R\rangle \in \text{Poss}(H')$. We now apply Axiom I to place $\langle \text{inv } A\rangle$ in P, and then apply Axiom C in sequence to each invocation in L but not in L', yielding a derivation that $\langle v, P', R'\rangle \in \text{Poss}(H)$.

5. An Extended Proof of Correctness

5.1. Main Proof

Figure 5-1 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 4.2.2. 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.

```
{∃ ⟨q, P, R⟩ ∈ Poss}
Enq = proc (q: queue, x: item)
{∃ ⟨q', P', R'⟩ ∈ Poss' . q' = q ∧ P' = P ∪ {Enq(x) A} ∧ R' = R}

    {∃ ⟨q, P, R⟩ ∈ Poss . ⟨Enq(x) A⟩ ∈ P}
    i: int := INC(q.back)
    {Poss' = Poss}

    {∃ ⟨q, P, R⟩ ∈ Poss . ⟨Enq(x) A⟩ ∈ P}
    STORE(q.items[i], x)
    {∃ ⟨q', P', R'⟩ ∈ Poss' . P' = P - {Enq(x) A} ∧ R' = R ∪ {Ok() A} ∧
          index(q.items', x) = i ∧ x ∈ max(items(q')) ∧ q.back ≤ q.back'}

    {∃ ⟨q, P, R⟩ ∈ Poss . ⟨Ok() A⟩ ∈ R}
    end Enq
    {∃ ⟨q', P', R'⟩ ∈ Poss' . q' = q ∧ P' = P ∧ R' = R - {Ok() A}}

{∃ ⟨q, P, R⟩ ∈ Poss}
Deq = proc (q: queue) returns (item)
{∃ ⟨q', P', R'⟩ ∈ Poss' . q' = q ∧ P' = P ∪ {Deq() A} ∧ R' = R}

    {∃ ⟨q, P, R⟩ ∈ Poss . ⟨Deq() A⟩ ∈ P}
    while true do
      range: int := READ(q.back)-1
      {Poss' = Poss}

      for i: int in 1..range do

        {∃ ⟨q, P, R⟩ ∈ Poss . ⟨Deq() A⟩ ∈ P}
          x: item := SWAP(q.items[i], null)
          {∃ ⟨q', P', R'⟩ ∈ Poss' . P' = P - {Deq() A} ∧ R' = R ∪ {Ok(x) A} ∧
             (x = null ∨ x ∈ min(items(q')))}

          if x ~= null then return(x) end
          end
        end
    end

    {∃ ⟨q, P, R⟩ ∈ Poss . ⟨Ok(x) A⟩ ∈ R}
    end Deq
    {∃ ⟨q', P', R'⟩ ∈ Poss' . q' = q ∧ P' = P ∧ R' = R - {Ok(x) A}}
```

Figure 5-1: Annotated Queue Implementation

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

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

and

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

The partially ordered set of queue items, (I, <), captures the non-quiescent 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 $A(r) = (\text{items}(r), <_r)$. The set [(I, <), P, R] identifies each of the possible sets of queue values with a set of pending invocations and a set of possible responses, thereby forming a set of (queue) possibilities. The following two lemmas make use of Lemma 17, proved in the Appendix.

Lemma 5: 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 $[(I \cup \{x\}, <), P, R] \subseteq Poss$.

Proof: Pick any $q \in (I, <)$, and any $q' \in (I \cup \{x\}, <)$. Since $\langle q, P \cup \{Enq(x)\ A\}, R - \{Ok()\ A\}\rangle \in Poss$, $\langle ins(q,x), P, R\rangle \in Poss$ by Axiom C. Since ins(q,x) is an element of $(I \cup \{x\}, <)$, $\langle q', P, R\rangle \in Poss$ by Lemma 17, where $q' = ins(q,x)$.

Lemma 6: 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, $[(I - \{x\}, <), P, R] \subseteq Poss$.

Proof: Pick any $q \in (I,<)$ such that $first(q) = x$, and any $q' \in (I - \{x\}, <)$. Since $\langle q, P \cup \{Deq()\ A\}, R - \{Ok(x)\ A\}\rangle \in Poss$, $\langle rest(q), P, R\rangle \in Poss$ by Axiom C. Since rest(q) is an element of $(I - \{x\}, <)$, $\langle q', P, R\rangle \in Poss$ by Lemma 17, where $q' = rest(q)$.

Lemma 5 will allow us to show that the set of linearized queue values does not change over a STORE operation and similarly, Lemma 6, 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 7: Enq enqueues an item x that is maximal with respect to $<_r$.

Proof: Suppose not. Then after the STORE there exists some non-null item y such that $x <_r y$. By definition of $<_r$, we have that the STORE for x precedes the INC for y. Thus, index(q.items, x) < index(q.items, y). Since index(q.items, x) = q.back, then q.back < index(q.items, y). By the rep invariant, for all i, $i \geq$ q.back, q.items[i] = null so that q.items[index(q.items, y)] = null, i.e., y = null, a contradiction.

Lemma 8: Deq removes and returns an item x that is minimal with respect to $<_r$.

Proof: Suppose not. Then there exists non-null y such that $y <_r x$. For x to be returned from within the for loop, the SWAP of x must happen before the STORE of y. The STORE of x must happen before the SWAP of x and the INC of x before the STORE of x, so then the INC of x must occur before the STORE of y, which implies that x and y are incomparable, a contradiction.

Here is a proof of correctness.

Theorem 9: The queue implementation is correct.

Proof: Assuming every rep history is linearizable, we need to show that every queue history, H | q, is linearizable. It suffices to show that the "subset" property, $\cup_{r\in Lin(H\ |\ r)}A(r) \subseteq Lin(H\ |\ q)$, 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 5-1 as justified by the Owicki-Gries proof method [15]. 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 7 we know that STORE adds a maximal item and thus, we can apply Lemma 5 to show that the subset property is preserved. Similarly, by Lemma 8 we know that SWAP removes a minimal item and thus, we can apply Lemma 6 to show that the subset property is preserved.

5.2. An Aside: Handling Critical Regions

An implementation without critical regions, such as the previous queue example, can be verified by defining a rep invariant that is continually satisfied, and an abstraction function that is continually defined. That is, each step of the sequence of representation operations implementing an abstract operation must preserve the rep invariant, and exactly one such step causes the operation's effects to become visible to other operations.

If an operation creates a temporary inconsistency, perhaps hidden from concurrent operations by some form of critical region, then it may not be possible to define a meaningful abstraction function directly in terms of the representation. Such inconsistencies can be eliminated by augmenting the representation with appropriate auxiliary data.

6. Reasoning About Concurrent Objects

We now 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. For verifying implementations of objects, i.e., program text, it suffices to reason simply in terms of sets of values, e.g., Lin(H) for some history H, as we did in the previous section, but as we illustrate below, sometimes it more convenient to reason in terms of sets of linearizations, i.e., sets of sequential histories. We use Theorem 3 and type-specific instantiations of Axioms C, I, R, and S to prove properties about concurrent objects. First, we look at concurrent registers, then concurrent queues.

6.1. Concurrent Registers

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

{true}
Read() / Ok(v)
{fetch(r) = fetch(r') = v}

{true}
Write(v) / Ok()
{fetch(r') = v}

where the Larch Shared Language specification for register values is:

```
RVals: trait
   introduces
      new: → R
      store: R, V → R
      fetch: R → V
      dontcare: → V
   constrains R so that for all r: R, v: V
      fetch(new) = dontcare
      fetch(store(r, v)) = v
```

These sequential axioms can be combined with our linearizability condition to prove assertions about the interleavings permitted by concurrent registers.

Every value read was written, but not overwritten.

Theorem 10: If the last event of H is the Read response ⟨Ok(v) A⟩, then H includes an earlier Write invocation ⟨Write(v) B⟩, and if the Write operation is complete, then it precedes no other complete Write operation.

Proof: If the Read response is the mth event of H, then $\langle v, P, R\rangle \in Poss_m$. In any derivation showing that H is linearizable, the last application of Axiom C for a Write invocation must have the form:

$\langle u, Q, S\rangle \in \text{Poss}_k \Rightarrow \langle v, Q - \{\text{Write}(v)\ B\}, S \cup \{\text{Ok}()\ B\}\rangle \in \text{Poss}_k$

This inference is legal only if B's Write is pending at event k. By Theorem 3, if B's Write is complete and precedes another complete Write, then the derivation must include a later application of Axiom C for a Write invocation, contradicting our assumption that B's was the last.

Register values are persistent in the absence of Write operations.

Theorem 11: 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 that lie within the interval that return distinct values v and v'. If H is linearizable, there exists a derivation showing that $\langle v, P, R\rangle \in \text{Poss}_j(H)$ and $\langle v', Q, S\rangle \in \text{Poss}_k(H)$ where one Read is pending at event j and the other at event k, where $j \leq k$. The only way to deduce that $\langle v', Q, S\rangle \in \text{Poss}_k(H)$ from $\langle v, P, R\rangle \in \text{Poss}_j(H)$ is to apply Axiom C to a pending Write at some intermediate step, which is permissible only if some Write operation overlaps *I*.

6.2. Concurrent Queues

The proofs of the following theorems about concurrent queues use the following fact about queues:

Lemma 12: If Q is a sequential queue history where x is enqueued before y, then x is not dequeued after y.

Proof: From Axioms E, D, and F of Figure 2-1.

Theorem 13: If [Enq(x)/Ok() P], [Enq(y)/Ok() Q], [Deq()/Ok(x). R], and [Deq()/Ok(y) S] 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: Pick a derivation showing H is linearizable. Theorem 3 implies that Axiom C is applied to all four invocations, since the operations are complete. Moreover, because the enqueue of x precedes the enqueue of y, the derivation must apply Axiom C to x's Enq first. By Lemma 12, the derivation must also apply Axiom C to x's Deq before y's Deq, thus y's Deq operation cannot precede x's Deq.

Gottlieb, Lubachevsky, and Rudolph [2] adopt the property proved in Theorem 13 as the (informal) correctness property for a linearizable queue implementation. The difficulty of reasoning informally about concurrent histories is illustrated by observing that Theorem 13 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 14: 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: Any derivation showing that H is linearizable must use Axiom C to enqueue x before enqueuing y, hence by Lemma 12 the derivation must apply Axiom C to dequeue x before it can dequeue y. The Deq invocation that removed x may have returned, or it may be pending.

Items do not spontaneously appear in the queue.

Theorem 15: If x has been dequeued, then it was enqueued, and the Deq operation does not precede the Enq.

Proof: Any derivation showing that $\langle q, P, R\rangle \in \mathrm{Poss}_k$ and $x = \mathrm{first}(q)$ must include an earlier application of Axiom C showing that:

$$\langle q', P', R'\rangle \in \mathrm{Poss}_j \Rightarrow \langle \mathrm{ins}(q',x), P' - \{\mathrm{Enq}(x)\ A\}, R' \cup \{\mathrm{Ok}()\ A\}\rangle \in \mathrm{Poss}_j$$

which is legal only if the invocation has occurred.

7. Final Remarks

7.1. Related Work

Our notion of linearizability generalizes and unifies similar notions found in specific examples in the literature. One of the earliest papers to verify a linearizable concurrent object is due to Lamport [12], who verifies a queue implementation that permits one enqueuing process to execute concurrently with one dequeuing process. His verification technique is based on a continually-defined abstraction function (called a state function) that maps the representation onto a single queue 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. As noted above, this technique is not powerful enough to verify highly concurrent objects such as the queue implementation given in Section 4.

Misra [14] 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 10 and 11 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.

7.2. Significance of Linearizability

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 (e.g., Lamport's [12]). 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 serializability for data base 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 [16, 12, 13], must be used to reason about liveness properties like 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.

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.

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.

References

[1] 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 of the ACM 19(11):624-633, November, 1976.

[2] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph.
Basic Techniques for the efficient coordination of very large numbers of cooperating sequential processors.
ACM Transactions on Programming Languages and Systems (2):164-189, April, 1983.

[3] J.V. Guttag, E. Horowitz, and D.R. Musser.
Abstract Data Types and Software Validation.
CACM 21(12):1048-1064, December, 1978.

[4] J.V. Guttag, J.J. Horning, and J.M. Wing.
Larch in Five Easy Pieces.
Technical Report 5, DEC Systems Research Center, July, 1985.

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

[6] M. Herlihy and J. Wing.
Axioms for concurrent objects.
In *14th ACM Symposium on Principles of Programming Languages*, pages 13-26. January, 1987.

[7] M.P. Herlihy.
Dynamic quorum adjustment for partitioned data.
ACM Transactions on Database Systems 12(2):170-194, June, 1987.

[8] M.P. Herlihy.
Impossibility and Universality Results for Wait-Free Synchronization.
In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC).* August, 1988.
To appear.

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

[10] C.P. Kruskal, L. Rudolph, and M. Snir.
Efficient synchronizaiton on multiprocessors with shared memory.
In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* August, 1986.
To appear.

[11] L. Lamport.
How to make a multiprocessor computer that correctly executes multiprocess programs.
IEEE Transactions on Computers C-28(9):690, September, 1979.

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

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

[14] J. Misra.
Axioms for Memory Access in Asynchronous Hardware Systems.
ACM Transactions on Programming Languages and Systems 8(1):142-153, January, 1986.

[15] S. Owicki and D. Gries.
An Axiomatic Proof Technique for Parallel Programs.
Acta Informatica 6(4):319-340, 1976.

[16] S. Owicki and L. Lamport.
Proving Liveness Properties of Concurrent Programs.
ACM Transactions on Programming Languages and Systems 4(3):455-495, July, 1982.

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

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

I. Two Lemmas about Concurrent Queues

In our verification of the queue implementation of Section 5 we used the following two lemmas about queues, which we prove below.

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

$$\langle q_j, P_j, R_j \rangle \in Poss_k$$

$$\Rightarrow \langle ins(q_j, x), P_j - \{Enq(x)\ A\}, R_j \cup \{Ok()\ A\}\rangle \in Poss_k$$

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 $\langle q, P, R\rangle \in Poss_m$ 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 16 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 16: If δ is a canonical derivation showing that ⟨q, P, R⟩ ∈ $Poss_m$, 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).

Proof: We show that x's Enq inference commutes with all other inferences. If the next inference in δ is the Deq inference for an item y, then δ cannot be canonical, because:

$\langle q_j, P_j, R_j\rangle \in Poss_k$
$\Rightarrow \langle ins(q_j,x), P_j - \{Enq(x)\ A\}, R_j \cup \{Ok()\ A\}\rangle \in Poss_k$
$\Rightarrow \langle rest(ins(q_j,x)), P_j - \{Enq(x)\ A, Deq()\ B\}, R_j \cup \{Ok()\ A, Ok(y)\ B\}\rangle \in Poss_k$

is equivalent to:

$\langle q_j, P_j, R_j\rangle \in Poss_k$
$\Rightarrow \langle rest(q_j)), P_j - \{Deq()\ B\}, R_j \cup \{Ok(y)\ B\}\rangle \in Poss_k$
$\Rightarrow \langle ins(rest(q_j),x), P_j - \{Enq(x)\ A, Deq()\ B\}, R_j \cup \{Ok()\ A, Ok(y)\ B\}\rangle \in Poss_k$

Here, we exploit the observation that because x is in q, q_j must be non-empty, hence $rest(ins(q_j,x)) = ins(rest(q_j),x)$.

Similar arguments show that x's Enq inference commutes with all applications of Axiom I, and with all applications of Axiom R for non-matching response events. Finally, we observe that any Enq inference for an item in q must follow all Enq inferences for items whose Deq inferences appear in δ.

Lemma 17 states that we can consider equivalence classes of queues rather than individual queues.

Lemma 17: If ⟨q, P, R⟩ ∈ $Poss_m$, 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 ⟨q*, P, R⟩ ∈ $Poss_m$.

Proof: We argue inductively that if there exists a canonical n-step derivation that ⟨q, P, R⟩ ∈ $Poss_m$, there also exists a canonical n-step derivation that ⟨q*, P, R⟩ ∈ $Poss_m$.

Base step: Trivial by Axiom S for a canonical derivation of length 0, where q = emp.

Induction hypothesis: If ⟨q, P, R⟩ ∈ $Poss_m$ has a canonical derivation of length less than n, ⟨q*, P, R⟩ ∈ $Poss_m$ has a canonical derivation of the same length.

Induction step: Given an n-step canonical derivation δ that ⟨q, P, R⟩ ∈ $Poss_m$, we construct an n-step canonical derivation δ* that ⟨q*, P, R⟩ ∈ $Poss_m$. If the last step of δ is an application of Axiom I or R, then $q_{n-1} = q_n$, and we have an n-1 step canonical derivation that $\langle q_n, P_{n-1}, R_{n-1}\rangle \in Poss_{m-1}$. The induction hypothesis yields an n-1 step canonical derivation that $\langle q^*, P_{n-1}, R_{n-1}\rangle \in Poss_{m-1}$, and reapplying the last inference yields a derivation that $(q^*, P_n, R_n) \in Poss_m$.

Otherwise, the last step of δ is an Enq or Deq inference, which can be discarded to yield an n-1 step canonical derivation that $\langle q_{n-1}, P_{n-1}, R_{n-1}\rangle \in Poss_m$. Suppose the discarded inference is an Enq inference for x by A. Define q_{n-1}^* to be q* with x deleted from the queue. By the induction hypothesis, there exists an n-1 step canonical derivation δ_{n-1}^* that $\langle q_{n-1}^*, P_{n-1}, R_{n-1}\rangle \in Poss_m$. If x is the last element in q*, then we construct δ* using Axiom C to enqueue x to q_{n-1}^*. Otherwise, let y be the item immediately following x in q*, let B be the process that enqueued y, and let the jth inference of δ_{n-1}^* be the Enq inference for y. By Lemma 16, the next event in the history is the return event for some item z that follows x in q*. Since z's Enq operation is concurrent with x's Enq operation, $\langle \mathrm{Enq}(x)\ A\rangle \in P_j^*$. We construct δ^* as follows: all inferences before j are unchanged, and the jth inference of δ^* is x's Enq inference:

$$\langle q_j^*, P_j^*, R_j^*\rangle \in \mathrm{Poss}_k$$
$$\Rightarrow \langle \mathrm{ins}(q_j^*,x), P_j^* - \{\mathrm{Enq}(x)\ A\}, R_j^* \cup \{\mathrm{Ok}()\ A\}\rangle \in \mathrm{Poss}_k$$

which is justified because $\langle \mathrm{Enq}(x)\ A\rangle$ is in P_j^*. For $j < k \leq n$, the kth inference of δ^* is the (k-1)st inference of δ, with $\mathrm{ins}(q_j^*,x)$ substituted for q_j^*, P_k^* for P_k, and R_k^* for R_k. To show that δ^* is sound, we must check that each axiom's pre-condition is still satisfied. The result is immediate for applications of Axioms I and R, as well as for Enq inferences, since it is always legal to append an Enq to a history. For Deq inferences, we observe that every dequeued item was enqueued before x, hence at each Deq inference, the value at the front of the queue is unchanged. Finally, δ^* is canonical because the Enq inferences for x and y do not commute.

Suppose the discarded inference was a Deq inference, where $\mathrm{first}(q_{n-1}) = x$. Define q_{n-1}^* to be the queue value such that $\mathrm{first}(q_{n-1}^*) = x$ and $\mathrm{rest}(q_{n-1}^*) = q^*$. By the induction hypothesis, there exists a canonical n-1 step derivation δ_{n-1}^* that $\langle q_{n-1}^*, P_{n-1}, R_{n-1}\rangle \in \mathrm{Poss}_m$. Since $\langle \mathrm{Deq}()\ A\rangle \in P_{n-1}$, we can use Axiom C to extend δ_{n-1}^* to a canonical derivation δ^* such that $\langle \mathrm{rest}(q_{n-1}^*), P_{n-1} - \{\mathrm{Deq}()\ A\}, R_{n-1} \cup \{\mathrm{Ok}(x)\ A\}\rangle = \langle q^*, P_n, R_n\rangle \in \mathrm{Poss}_m$.