CMU-CS-86-154

Axioms for Concurrent Objects

Maurice P. Herlihy
(Herlihy@C.CS.CMU.EDU)

Jeannette M. Wing
(Wing@C.CS.CMU.EDU)

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

24 October 1986

Abstract

Specification and verification techniques for abstract data types that have been successful for sequential programs can be extended in a natural way to provide the same benefits for concurrent programs. We propose an approach to specifying and verifying concurrent objects based on a novel correctness condition, which we call *linearizability*. 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. In this paper, we will define and discuss linearizability, and then give examples of how to reason about concurrent objects and verify their implementations based on their (sequential) axiomatic specifications.

This paper will appear in the Proceedings of the Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), Munich, West Germany, January 21-23, 1987.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J.M. Wing was provided in part by the National Science Foundation under grant DMC-8519254. 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.

1. Introduction

This paper shows that the specification and verification techniques for abstract data types that have been successful for sequential programs can be extended in a natural way to provide the same benefits for concurrent programs. Our two main contributions are:

- New techniques for using (sequential) axiomatic specifications to reason about concurrent objects; and
- A novel correctness condition, which we call *linearizability*.

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.

Our approach to specifying and verifying concurrent objects is based on the notion of *linearizability*. 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 [17] or serializability [26], 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 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 show that we can perform two kinds of reasoning:

- We reason about concurrent computations by transforming assertions about concurrent computations into simpler assertions about sequential computations. Familiar axiomatic techniques help prove these transformed assertions.

- Implementations of concurrent objects are necessarily more complex then their sequential counterparts. We reason about the correctness of linearizable implementations using new techniques that generalize the notions of representation invariant and abstraction function to the concurrent domain.

Section 2 presents our model of a concurrent system and specification technique; Section 3 defines and discusses linearizability, including its locality property; Section 4 illustrates reasoning about concurrent registers and queues; Section 5 illustrates reasoning about an implementation of a concurrent queue; Sections 6 and 7 contain discussions on related work and the significance of linearizability.

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:

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). 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.

For example, let H_1 be the following history:

(History H_1)

```
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
```

H_1 is a well-formed history for a FIFO queue q providing Enq and Deq operations. The first event 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) B/q Ok() B" operation lies within the "q Enq(x) A/q 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 *axiomatic specifications* to reason about object values. A specification is a set of axioms of the form:

```
{P}
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 [10, 11] 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. 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 object. The set of operators and their signatures following **introduces** defines a vocabulary of

Axiom E:

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

Axiom D:

{¬ isEmp(q)}
Deq()/Ok(e)
{q' = rest(q) ∧ e = 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

terms to 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 terms of sort Q. 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

3.1. Definition

Axiomatic specifications have (as yet) no meaning for histories that are not sequential. This section introduces the notion of *linearizability*, the basic correctness condition that allows us to apply algebraic specifications and axiomatic reasoning to concurrent objects.

A history H induces an irreflexive partial order $\prec_H$ on operations:

$e_0 \prec_H e_1$ if $res(e_0)$ precedes $inv(e_1)$ in H.

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

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

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

L2: $\prec_{H'} \subseteq \prec_S$.

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.

The history H_1 shown in Section 2 is linearizable, because $H_1' = H_1 \bullet$ q Ok() A is equivalent to the following sequential history:

(History H_1')

```
q Enq(x) A
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
```

The following history, H_2, is not linearizable:

(History H_2)

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

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

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

q Enq(x) A
q Deq() B
q Ok(x) B (History H_3)

H_3 can be extended to $H_3' = H_3 \bullet$ q Ok() A, which can be shown equivalent to the sequential history in which the enqueue operation occurs before the dequeue.

Lamport's notion of *sequential consistency* [17] requires that a history be equivalent to a sequential history. Sequential consistency is weaker than linearizability, because it does not require the precedence ordering $\prec$ to be preserved. For example, H_2 is sequentially consistent, but not linearizable. *Serializability* [26] requires that a history be equivalent to a sequential history in which each process (usually called a transaction) runs to completion without interleaving with other processes. [1] Lamport [19] has proposed that the standard definition of serializability be strengthened to preserve the order of non-overlapping transactions. Both notions of serializability are incomparable to linearizability. For example, H_1 is linearizable, but not serializable (in either sense), and H_2 is serializable, but not linearizable.

3.2. 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, $\prec$, and hence can be extended to a total order, $<$. Notice that each $<_x$ and $<_P$ is compatible with $\prec$.

Suppose $\prec$ is not a partial order. Then we can construct a cycle $e_1 \bullet \ldots \bullet e_n$, where $e_1 = e_n$, such that $e_1 \prec e_2 \prec \ldots \prec e_n$, 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, the cycle of operations can be relabeled so 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 $\prec$, 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 consider only histories involving single objects, omitting object names from events.

Locality is important because it allows concurrent systems to be designed and constructed in a

[1] In databases, serializability is often provided in conjunction with *failure atomicity*, ensuring that a transaction interrupted by a failure will have no effect.

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 additional constraints must be placed on objects to ensure that they follow compatible scheduling protocols.

Locality should not be taken for granted; the literature includes proposals for both alternative correctness properties that are not local. For example, Lamport's notion of sequential consistency is not a local property. Consider the following sequential history H, in which processes A and B operate on queue objects p and q. For brevity, matching responses are shown on the same lines as invocations.

p Enq(x)/Ok() A
q Enq(y)/Ok() B
q Enq(x)/Ok() A
p Enq(y)/Ok() B
p Deq()/Ok(y) A
q Deq()/Ok(x) B

H|p and H|q are not legal sequential histories, but they are sequentially consistent, although H itself is not.

Also, serializability and Lamport's strengthened notion of serializability are both non-local. In the example above, H|p and H|q are each serializable in either sense, but H itself is not.

3.3. Linearized Values

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. We call the value of an object at the end of a linearization 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.

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. (We use [] for emp and [x,y] for ins(ins(emp, x),y), etc.) 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.

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

4. Reasoning About Concurrent Objects

So far, linearizability is defined 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. Sometimes it will be necessary or more convenient to reason in terms of sets of sequential histories, (see Poss below), and sometimes, simply in terms of sets of values, e.g., Lin(H) for some history H. In this section, we show how we reason about properties of concurrent objects by reasoning in terms of sets of linearizations; in the next section, we how how we do verification by reasoning in terms of sets of values.

A *possibility* for a history H is a pair <S, P> where S is a linearization of H and P is the set of pending invocations *not* completed to construct S. We let Poss denote the set of possibilities of a history. If we only care about the final value of S, we use <v, P> to denote one of the <S, P> such that the value of S is v. The relationship between the set of possibilities and set of linearized values for a given history H is that for each <v, P> $\in$ Poss, v $\in$ Lin(H).

H's set of possibilities, and hence set of linearized values, is captured by the following three axioms. The following *closure axiom* states that if S is a linearization of H, 'inv A' is a pending invocation in H that is not completed to form S, and S' = S • inv A • res A is a legal sequential history, then S' is also a linearization of H.

Axiom C:
<S, P> $\in$ Poss
$\Rightarrow$ [($\forall$ 'inv A' $\in$ P) ($\exists$ res) S • inv A • res A is legal
$\Rightarrow$ <S • inv A • res A, P – {inv A}> $\in$ Poss]

The following *invocation axiom* states that any linearization of H is also a linearization of H • inv A.

Axiom I:

$$\{\langle S, P\rangle \in Poss\}$$
$$\text{inv } A$$
$$\{\langle S, P \cup \{\text{inv } A\}\rangle \in Poss'\}$$

Let last(S, A) be the response to A's last invocation in the sequential history S, and let $A \in P$ mean there exists an invocation i such that 'i A' $\in$ P. 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:

$$\{\langle S, P\rangle \in Poss \text{ and } A \notin P \text{ and res } A = last(S, A)\}$$
$$\text{res } A$$
$$\{\langle S, P\rangle \in Poss'\}$$

For a given history with m events, we use $Poss_i$ to denote the set of possibilities for the ith prefix of H, for $0 \leq i \leq m$. A *derivation* that shows that $\langle v, P\rangle \in Poss_m$ is a sequence of implications of the form:

$\langle v_0, P_0\rangle \in Poss_0$
$\Rightarrow$ ...
$\Rightarrow \langle v_j, P_j\rangle \in Poss_k$
$\Rightarrow$ ...
$\Rightarrow \langle v_n, P_n\rangle \in Poss_m$.

where $v_n = v$, $P_n = P$, and each implication is justified by Axiom C, I, or R.

Intuitively, a derivation is like a history. Instead of reasoning about histories directly, however, we use axiomatic proof techniques to reason about derivations—each implication in a derivation is like a step in a proof where each step in the proof is justified by some axiom. For each operation of a typed object, Axiom C is instantiated to yield type-specific closure axioms, and similarly for Axioms I and R.

In any derivation showing H is linearizable, the order in which Axiom C is applied to pending invocations induces a valid linearization ordering on the operations of H. Informally, the following lemma states that an operation must appear to "take effect" at some instant between its invocation and its response.

Lemma 2: Let 'inv A' be the ith event of H, and let the matching response 'res A' be the jth event. Any derivation showing that H is linearizable must include an application of Axiom C to infer:

$$\langle v, P\rangle \in Poss_k \Rightarrow \langle v', P - \{\text{inv } A\}\rangle \in Poss_k$$

for some k, $i \leq k < j$.

Proof: The only way to infer anything about $Poss_i$ from $Poss_{i-1}$ is to apply Axiom I as follows:

$$\langle u, Q\rangle \in Poss_{i-1} \Rightarrow \langle u, Q \cup \{\text{inv } A\}\rangle \in Poss_i$$

Later in the derivation, the only way to infer anything about $Poss_j$ from $Poss_{j-1}$ is by applying Axiom R:

$\langle w, R\rangle \in Poss_{j-1}$ and 'inv A' $\notin R \Rightarrow \langle w, R\rangle \in Poss_j$.

Between these two steps, the only way to remove 'inv A' from the set of pending invocations is by applying Axiom C as shown above. ∎

We use Lemma 2 and type-specific instantiations of Axioms C, I, and R to prove properties about concurrent objects. First, we look at concurrent registers, then concurrent queues.

4.1. 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. Notice that we do not need to assume that all values written to the register are unique.

Every value read was written, but not overwritten.

Theorem 3: 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 $\langle v, P\rangle \in Poss_m$. In any derivation showing that H is linearizable, the last application of Axiom C for a Write invocation must have the form:

$\langle u, Q\rangle \in Poss_k \Rightarrow \langle v, Q - \{\text{Write(v) B}\}\rangle \in Poss_k$

This inference is legal only if B's Write is pending at event k. By Lemma 2, 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 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 that lie within the interval that return distinct values v and v'. If H is linearizable, there exists a derivation showing that $\langle v, P\rangle \in Poss_j$ and $\langle v', Q\rangle \in Poss_k$ where one Read is pending at event j and the other at event k, where $j \leq k$. The only way to deduce that $\langle v', Q\rangle \in Poss_k$ from $\langle v, P\rangle \in Poss_j$ is to apply Axiom C to a pending Write at some intermediate step, which is permissible only if some Write operation overlaps *I*. ∎

4.2. Concurrent Queues

The proofs of the following theorems about concurrent queues use Axioms E and D of Figure 2-1 and the trait of Figure 2-2. We make use of the following fact about queues:

Lemma 5: If Q is a sequential queue history where x is enqueued before y, then x is not dequeued after y.

Proof: From Axioms E and D. ∎

Theorem 6: If Enq(x)/Ok(), Enq(y)/Ok(), Deq()/Ok(x), and Deq()/Ok(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: Pick a derivation showing H is linearizable. Lemma 2 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 5, 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 et al. [7] adopt the property proved in Theorem 6 as the desired correctness property for their concurrent queue implementation. The difficulty of reasoning directly about interleaved 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: Any derivation showing that H is linearizable must use Axiom C to enqueue *x* before enqueuing *y*, hence by Lemma 5 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 8: If x has been dequeued, then it was enqueued, and the Deq operation does not precede the Enq.

Proof: Any derivation showing that $(q, P) \in Poss_k$ and $x = first(q)$ must include an earlier application of Axiom C showing that:

$$\langle q', P' \rangle \in Poss_j \Rightarrow \langle ins(q', x), P' - \{Enq(x)\ A\} \rangle \in Poss_j$$

which is legal only if the invocation has occurred. ∎

4.2.1. Two Lemmas about Concurrent Queues

Before we turn to verification of implementations, we state and prove two lemmas about queues that will be used to help verify the queue implementation of the next section.

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

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

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 that $\langle q, P \rangle \in Poss_m$ 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 9 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 9: If δ is a canonical derivation showing that $\langle q, P \rangle \in Poss_m$, 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).

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:

$$\begin{aligned} &\langle q_j, P_j \rangle \in Poss_k \\ &\Rightarrow \langle ins(q_j, x), P_j - \{Enq(x)\ A\} \rangle \in Poss_k \\ &\Rightarrow \langle rest(ins(q_j, x)), P_j - \{Enq(x)\ A, Deq()\ B\} \rangle \in Poss_k \end{aligned}$$

is equivalent to:

$$\begin{aligned} &\langle q_j, P_j \rangle \in Poss_k \\ &\Rightarrow \langle rest(q_j)), P_j - \{Deq()\ B\} \rangle \in Poss_k \\ &\Rightarrow \langle ins(rest(q_j), x), P_j - \{Enq(x)\ A, Deq()\ B\} \rangle \in Poss_k \end{aligned}$$

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

Lemma 10: If $\langle q, P\rangle \in Poss_m$, 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 $\langle q^*, P\rangle \in Poss_m$.

Proof: We argue inductively that if there exists a canonical n-step derivation that $\langle q, P\rangle \in Poss_m$, there also exists a canonical n-step derivation that $\langle q^*, P\rangle \in Poss_m$.

Base step: Trivial for $n = 0$ where $q = emp$.

Induction hypothesis: If $\langle q, P\rangle \in Poss_m$ has a canonical derivation of length less than n, $\langle q^*, P\rangle \in Poss_m$ has a canonical derivation of the same length.

Induction step: Given an n-step canonical derivation δ that $\langle q, P\rangle \in Poss_m$, we construct an n-step canonical derivation δ^* that $\langle q^*, P\rangle \in Poss_m$. 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 $\langle q_n, P_{n-1}\rangle \in Poss_{m-1}$. The induction hypothesis yields an n-1 step canonical derivation that $\langle q^*, P_{n-1}\rangle \in Poss_{m-1}$, and reapplying the last inference yields a derivation that $(q^*, P_n) \in Poss_m$.

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 $\langle q_{n-1}, P_{n-1}\rangle \in Poss_m$. 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 $\delta_{n-1}{}^*$ that $\langle q_{n-1}{}^*, P_{n-1}\rangle \in Poss_m$. 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 $\delta_{n-1}{}^*$ be the Enq inference for y. By Lemma 9, 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, 'Enq(x) A' $\in P_j{}^*$. We construct δ^* as follows: all inferences before j are unchanged, and the jth inference of δ^* is x's Enq inference:

$$\langle q_j{}^*, P_j{}^*\rangle \in Poss_k$$
$$\Rightarrow \langle ins(q_j{}^*, x), P_j{}^* - \{Enq(x)\ A\}\rangle \in Poss_k$$

which is justified because 'Enq(x) A' is in $P_j{}^*$. For $j < k \leq n$, the kth inference of δ^* is the (k-1)st inference of δ, with $ins(q_j{}^*, x)$ substituted for $q_j{}^*$ and $P_k{}^*$ for P_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 $first(q_{n-1}) = x$. Define $q_{n-1}{}^*$ to be the queue value such that $first(q_{n-1}{}^*) = x$ and $rest(q_{n-1}{}^*) = q^*$. By the induction hypothesis, there exists a canonical n-1 step derivation $\delta_{n-1}{}^*$ that $\langle q_{n-1}{}^*, P_{n-1}\rangle \in Poss_m$. Since 'Deq() A' $\in P_{n-1}$, we can use Axiom C to extend $\delta_{n-1}{}^*$ to a canonical derivation δ^* such that $\langle rest(q_{n-1}{}^*), P_{n-1} - \{Deq()\ A\}\rangle = \langle q^*, P_n\rangle \in Poss_m$. ∎

5. Verification: Abstraction Functions Revisited

This section proposes a verification methodology for implementations of linearizable concurrent objects. We propose a model for data type implementations and define a correctness condition in Section 5.1. We give an example of a queue implementation and prove its correctness in Section 5.2.

As a first step, let us review how to verify a sequential data type implementation [14, 9]. 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*, $\mathcal{I}$: R → bool. The meaning of a legal rep is given by an *abstraction function*, $\mathcal{A}$: R → A, defined only for 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 exists a rep invariant, $\mathcal{I}$, and abstraction function, $\mathcal{A}$, such that whenever ρ carries one legal rep value r to another r', α carries the abstract value from $\mathcal{A}$(r) to $\mathcal{A}$(r').

For sequential objects, the rep invariant must hold at the start and finish of each abstract operation, but it may be violated 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. Instead, operations must be implemented to behave correctly for rep values that reflect the incomplete effects of concurrent operations. To capture the effects of interleaving, the notions of rep invariant and abstraction function must be extended to encompass transient representation values reflecting the incomplete effects of concurrent operations.

5.1. Definitions

An *implementation* is a set of histories in which events of two objects, a *representation* object r of type R and an *abstract* object a of type A, are interleaved in a constrained way: for each history H in the implementation, (1) the subhistories H|r and H|a satisfy the usual well-formedness conditions; and (2) for each 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|a is linearizable.

The rep invariant $\mathcal{I}$ must be continually satisfied and the abstraction function continually defined not

only between abstract operations, but also between each step in the sequence of rep operations implementing an abstract operation. The non-determinism inherent in a concurrent computation gives our abstraction functions a different flavor from their sequential counterparts. We redefine the abstraction function so that it maps each rep value to a (non-empty) set of abstract values:

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

To show correctness, the verification technique for sequential implementations is generalized as follows. Assume that the implementation of r is correct, hence H|r is linearizable for all H in the implementation. Let Lin(H|x) be the set of linearized values for x in H. Our verification technique focuses on showing the following property:

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

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

5.2. Queue Example

Consider the following concurrent queue implementation. The 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: cvt, x: item)
      i: int := INC(q.back)          % Allocate a new slot.
      STORE(q.items[i], x)           % Fill it.
      end enq

Deq = proc (q: cvt) returns (item)
      while true do
          range: int := FETCH(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 atomic steps: an array slot is reserved by atomically incrementing

back, and the new item is stored in *items*. [2] 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 *back-1* without encountering a non-null element, the operation is restarted. All atomic steps can be interleaved with steps of other operations. For brevity, we leave out the axioms and traits for records and arrays, which can be straightforwardly given (see [20, 10]).

Let R be a complete history for a queue representation, and let items(R) be the set of items stored in the array, but not swapped out. Let $\prec_R$ be the partial order such that $x \prec_R y$ if the STORE operation for x precedes the INC operation for y in R. If r is a linearized value for R, items(r) = items(R) corresponds to the set of non-null items in the array, and $\prec_r = \prec_R$ is their partial order. Finally, we extend the trait of Figure 2-2 by defining the total order, $<_q$, and the operator, items, such that:

first(q) $<_q$ first(rest(q))
items(emp) = {}
items(ins(q, e)) = {e} ∪ items(q)

The implementation has the following rep invariant:

$\mathcal{I}(r)$ = (r.back $\geq$ 1)
∧ (i $\geq$ r.back ⇒ r.items[i] = null)
∧ (lbound(r.items) = 1)

and the following abstraction function:

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

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 5-1 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.

Figure 5-2 shows the Enq and Deq implementation annotated with assertions that are true before and after each abstract invocation and response and each rep operation. It is convenient to keep as implicit auxiliary data the partial order, $\prec_R$, on items in the array, and the set, P, that contains the invocations that have not yet been applied to the rep.

---

[2] INC returns the value of its argument from before the invocation, not the new incremented value.

History	$\cup_{r \in Lin(H\|r)} \mathcal{A}(r)$	Lin(H|a)
		{[]}
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]}
FETCH(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 5-1: A Queue History

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

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

and

$$\langle (I, \prec), P \rangle = \{\langle q, P \rangle \mid q \in (I, \prec)\}$$

The partially ordered set of queue items, $(I, \prec)$, 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 $\mathcal{A}(r) = (items(r), \prec_r)$. $\langle (I, \prec), P \rangle$ identifies each of the possible sets of queue values with a set of pending invocations, thereby forming a set of (queue) possibilities.

Lemma 11: If x is a maximal element with respect to $\prec$, $x \notin I$, 'Enq(x) A' $\notin P$, and $\langle (I, \prec), P \cup \{Enq(x)\ A\} \rangle \subseteq$ Poss, then $\langle (I \cup \{x\}, \prec), P \rangle \subseteq$ Poss.

Proof: Pick any $q \in (I, \prec)$, and any $q' \in (I \cup \{x\}, \prec)$. Since $\langle q, P \cup \{Enq(x)\ A\} \rangle \in$ Poss, $\langle ins(q,x), P \rangle \in$ Poss by Axiom C. Since ins(q,x) is an element of $(I \cup \{x\}, \prec)$, $\langle q', P \rangle \in$ Poss by Lemma 10. ∎

Lemma 12: If $\langle (I, \prec), P \cup \{Deq()\ A\} \rangle \subseteq$ Poss, then for all x such that x is a minimal element of I, $\langle (I - \{x\}, \prec), P \rangle \subseteq$ Poss.

Proof: Pick any $q \in (I, \prec)$ such that first(q) = x, and any $q' \in (I - \{x\}, \prec)$. Since $\langle q, P \cup \{Deq()\ A\} \rangle \in$ Poss, $\langle rest(q), P \rangle \in$ Poss by Axiom C. Since rest(q) is an element of $(I - \{x\}, \prec)$, $\langle q', P \rangle \in$ Poss by Lemma 10. ∎

```
{true}
Enq = proc (q: cvt, x: item)
{P' = P ∪ {Enq(x) A}}

    {true}
    i: int := INC(q.back)
    {q.back' = q.back + 1 ∧ i = q.back}

    {'Enq(x) A' ∈ P}
    STORE(q.items[i], x)
    {P' = P - {Enq(x) A} ∧ index(q.items', x) = i ∧
       x ∈ max(items(q')) ∧ q.back ≤ q.back'}
         % A concurrent Enq might bump q.back before the store.

    {true}
    end Enq
    {'Enq(x) A' ∉ P'}

{true}
Deq = proc (q: cvt) returns (item)
{P' = P ∪ {Deq() A}}

    while true do

        {true}
        range: int := FETCH(q.back)-1
        {range = q.back-1}

        for i: int in 1..range do

            {true}
            x: item := SWAP(q.items[i],null)
            P' = P - {Deq() A} ∧ (x = null ∨ x ∈ min(items(q')))}

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

        end
    end Deq
```

Figure 5-2: Annotated Queue Implementation

Lemma 11 will allow us to show that the set of linearized queue values does not change over a STORE operation and similarly, Lemma 12, for a SWAP operation, by using $\prec_r$ for $\prec$ and by recalling that for each <v, P> $\in$ Poss, v is a linearized value. We use the next two lemmas to satisfy the conditions of the previous two lemmas.

Lemma 13: Enq enqueues an item x that is maximal with respect to $\prec_r$.

Proof: Suppose not. Then after the STORE there exists some non-null item y such that x $\prec_r$ y. By definition of $\prec_r$, 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 14: Deq removes and returns an item x that is minimal with respect to $\prec_r$.

Proof: Suppose not. Then there exists non-null y such that y $\prec_r$ 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 15: 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, $\bigcup_{r \in Lin(H|r)} \mathcal{A}(r) \subseteq Lin(H|q)$, 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 5-2 as justified by the Owicki-Gries proof method [23]. Axioms I and R give us the result for abstract invocation and response events. INC and FETCH leave the abstraction function the same. Thus, we are left with two cases, STORE and SWAP. By Lemma 13 we know that STORE adds a maximal item and thus, we can apply Lemma 11 to show that the subset property is preserved. Similarly, by Lemma 14 we know that SWAP removes a minimal item and thus, we can apply Lemma 12 to show that the subset property is preserved.

Proofs of non-interference between pre- and post-conditions and that the rep invariant holds are straightforward. ∎

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, and similar proof techniques as used in our queue example can be used for verification.

6. Related Work

The axiomatic approach to specifying sequential programs has its origins in Hoare's early work on verification [13] and later work on proofs of correctness of implementations of abstract data types [14], where first-order predicate logic pre- and post-conditions are used for the specification of each operation of the type. The algebraic approach, which defines data types to be heterogeneous algebras [3], 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 [1, 6, 8, 29, 4, 5, 28, 16]; we use more recent work on Larch specifications [10, 11] for sequential program modules.

Owicki and Gries extended Hoare's axiomatic approach to handle concurrent programs [24] by including axioms for general concurrent programming language constructs. Apt et al. [2] use a similar approach for CSP [15] in particular. These approaches differ from ours by focusing on control structures such as the parallel operator, leaving data uninterpreted.

Lamport [18] has proposed a model and assertion language for specifying properties of concurrent objects. His approach is more general than ours, as it addresses liveness as well as safety properties, and non-linearizable 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.

Our notion of linearizability generalizes and unifies similar notions found in specific examples in the literature. Lamport gives a specification for a linearizable concurrent queue permitting one enqueuing process and one dequeuing process. The queue's state is defined as a collection of *state functions* mapping time to algebraic values. One state function takes on queue values; it may change only while operations are in progress. The values of the other state functions are used as control flags to prevent operations from taking effect more than once. His queue-valued state function roughly corresponds to our abstraction function except that the state function maps to a single queue value, not a set of queue values. His technique, therefore, could not be used to prove our queue implementation correct because of the inherent nondeterminism in our example.

Misra [22] 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. [7] have investigated architectural support for implementing concurrent objects without critical regions, 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.

7. Discussion

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 [18]). 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 [27, 25, 12, 18, 21], 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 David Detlefs, Jim Horning, Leslie Lamport, Larry Rudolph, and William Weihl for lively verbal and electronic discussions about our notions of linearizability and correctness, and for their feedback on our extended abstract.

References

[1] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Abstract Data Types as Initial Algebras and Correctness of Data Representations. In *Proceedings from the Conference of Computer Graphics, Pattern Recognition and Data Structures*, pages 89-93. ACM, May, 1975.

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

[3] G. Birkhoff and J.D. Lipson. Heterogeneous Algebras. *Journal of Combinatorial Theory* 8:115-133, 1970.

[4] R.M. Burstall and J.A. Goguen. Putting Theories Together to Make Specifications. In *Fifth International Joint Conference on Artificial Intelligence*, pages 1045-1058. August, 1977. Invited paper.

[5] H.-D. Ehrich. Extensions and Implementations of Abstract Data Type Specifications. *Lecture Notes in Computer Science.* Volume 64.*Mathematical Foundations of Computer Science 1978 Proceedings.* Springer-Verlag, Poland, 1978, pages 155-164. 7th Symposium.

[6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[7] 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* 5(2):164-189, April, 1983.

[8] J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, Toronto, Canada, September, 1975.

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

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

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

[12] B.T. Hailpern and S. Owicki. Verifying Network Protocols Using Temporal Logic. In *Proceedings Trends and Applications 1980: Computer Network Protocols*, pages 18-28. IEEE Computer Society, 1980.

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

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

[15] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21(8):666-677, August, 1978.

[16] S. Kamin. Final Data Types and Their Specification. *ACM Transactions on Programming Languages and Systems* 5(1):97-121, January, 1983.

[17] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers* C-28(9):690, September, 1979.

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

[19] L. Lamport. *On Interprocess Communication*. Technical Report 8, DEC Systems Research Center, 1985. To appear in Distributed Computing.

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

[21] 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.

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

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

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

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

[26] C.H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM* 26(4):631-653, October, 1979.

[27] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46-57. Providence, RI, Oct. 31 - Nov. 2, 1977.

[28] M. Wand. Final Algebra Semantics and Data Type Extensions. *Journal of Computer and System Sciences* 19(1):27-44, August, 1979.

[29] S.N. Zilles. Abstract Specifications for Data Types. IBM Research Laboratory, San Jose, CA, 1975.