JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 17, 164–182 (1993)

Testing and Verifying Concurrent Objects

JEANNETTE M. WING

School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213

AND

CHUN GONG*

Department of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania 15260

A *concurrent object* is a data structure shared by concurrent processes. We present a two-pronged approach for establishing the correctness of implementations of concurrent objects. Our approach is based on a notion of correctness called *linearizability*: each concurrent object has a sequential specification, which describes how it behaves in sequential executions. Each concurrent execution is required to be equivalent to some sequential execution. We advocate using both testing and verification as complementary approaches for showing a concurrent object is linearizable. We first describe a simulation environment for simulating, testing, and analyzing implementations of concurrent objects. We can use the simulator to gain assurance that an implementation is correct or to detect errors in an incorrect one. The simulator provides a systematic way to testing implementations of any data type. Whereas testing is useful in practice, verification is more definitive. We provide a library of verified concurrent objects; specifically, we give the specifications, implementations, and proofs of correctness for the FIFO queue, semiqueue, and stuttering queue data types. Implementors of objects of other data types can use our library by following the patterns of design and proof that we give; at the same time, clients of our library need not reimplement commonly used abstractions from scratch and are freed from the tedium of their verification.

1. INTRODUCTION

A *concurrent system* is a collection of sequential threads of control called *processes* that communicate through shared data structures called *concurrent objects*. A concurrent object provides a finite set of primitive *operations* that are the only means to manipulate the object. Since an object's operations can be invoked by concurrent processes it is necessary to give meaning to possible interleavings of operation executions. Hence, given this model of concurrency, a fundamental question that arises is

- What is a reasonable notion of correctness for a system composed of concurrent objects?

While there is no general agreement on an answer to this question, we choose the correctness condition called *linearizability*, which has recently captured the attention of the research community. Linearizability, first coined in Herlihy and Wing's 1987 POPL paper [15], generalizes correctness notions that had previously been defined for specific data structures like atomic registers [23] and FIFO queues [8]. Informally, an execution in a concurrent system is *linearizable* if it is "equivalent," in a sense formally defined in Section 2, to a legal sequential execution. Linearizability implies that processes appear to be interleaved at the granularity of complete operations, and that the order of nonoverlapping operations is preserved [13].

Unlike alternative correctness conditions such as sequential consistency [19] and serializability [24], linearizability enjoys other properties like locality, which simplify the proof method. The locality property means that an entire system composed of individual linearizable objects is itself linearizable. The task of implementing a "correct" system is then simplified to that of implementing individual objects correctly.

A simple approach to implementing a concurrent object and guaranteeing that it is linearizable is to use critical regions [9], letting only a single process access the object at a time. However, critical regions unnecessarily limit the degree of concurrency possible when the type semantics of the object are ignored [15]. For example, multiple processes wishing to insert elements into a multiset should be permitted to go on concurrently without one blocking any of the others. Moreover, critical regions are ill-suited for asynchronous, fault-tolerant systems: if a faulty process halts in a critical region, nonfaulty processes will also be unable to progress [12].

Recently, other approaches for implementing linearizable objects have been proposed [14, 15, 3, 18–20, 22, 4,

* Work done while visiting Carnegie Mellon.

0734-7315/93 $5.00

7]. These approaches do not necessarily rely on using mutual exclusion locks to ensure the global consistency of a system composed of concurrent objects. However, because multiple processes can be simultaneously accessing an object, these implementations are typically tricky to reason about and hard to "get right." Thus, we are faced with a methodological problem:

- Given some notion of correctness, how can we show a given implementation is correct?

In particular, given linearizability as our notion of correctness, how can we show an implementation that does not rely on critical regions is correct?

We advocate using the complementary techniques of testing and verification. Rather than insist on using one technique to the exclusion of the other, we recognize the practical and formal benefits from using both validation techniques. Testing can help users detect when an implementation is incorrect and gain some additional assurance that an implementation is correct. Testing is especially useful in the debugging process while developing a correct implementation. Whereas testing is useful, verification is more definitive. "Program testing can be used to show the presence of bugs, but never to show their absences (Dijkstra [6])." In principle, a verified implementation is guaranteed to work in general rather than on a finite set of test cases. Verification, however, is often a tedious and difficult task.

Testing complements verification since verification often makes assumptions about the environment in which an implementation runs; testing helps to validate those assumptions. On the other hand, verification complements testing since it shows that an implementation is correct for any value of each its parameters, not just a finite number of values. Given our two-pronged approach to the methodological problem of showing correctness, the purpose of this paper is to report on the following new results:

1. A simulator useful for testing implementations of concurrent objects. More specifically, we describe the structure and use of a simulation environment in Section 3 and then the key algorithm that performs the simulation analysis in Section 4.
2. A library of verified concurrent objects. More specifically, we present the specifications, implementations, and proofs of correctness of some of the more interesting objects in our library: FIFO queues in Section 6, semiqueues in Section 7, and stuttering queues in Section 8.

The availability of both the simulator and library has pragmatic consequences for implementors and clients of linearizable objects. The simulator is a general-purpose testing tool. It is parameterized over the type of the data object that has been implemented and being tested. The simulator can automatically generate a set of test cases, saving the implementor from having to build a set of test cases for each implementation being tested. At the same, it also lets clients test out user-specified histories.

Proving linearizability of a data object can be a nontrivial task. In [20], the proof of a simple concurrent set consists of five propositions, one lemma, and one theorem. All our proofs are structured similarly. Thus, implementors of objects of data types other than those we provide in our library can reuse the patterns of design and proof that we give for our implementations. At the same time, people can use our library of objects without regard to their implementation and with the assurance that they are correct. Clients of our library are thus freed from having to design, implement, and verify commonly used abstractions from scratch. Indeed, one of our side goals is to encourage code reuse through program libraries.

First we begin in Section 2 to define our terminology, in particular the definition of linearizability. We adopt the notations and definitions as defined in [16], which contains a lengthier discussion of linearizability and motivating examples.

2. CONCURRENT OBJECTS AND LINEARIZABILITY

2.1. Model of Computation

Recall that a *concurrent system* is a collection of processes that communicate through shared data structures called *concurrent objects*, each providing a set of *operations*. Processes are sequential: each process applies a sequence of operations to objects, alternately issuing an invocation and receiving the associated response. Several processes might issue an invocation to the same object concurrently.

Formally, we model an execution of a concurrent system by a *history*, which is a finite sequence of operation invocation and response events. An operation invocation event is written as $x\ op(args^*)\ A$, where x is an object name, op is an operation name, $args^*$ is 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 the (normal or exceptional) termination condition and res^* is a sequence of result values. We use "Ok" for normal termination. A response matches an invocation if their object names agree and their process names agree. An invocation is pending in a history if no matching response follows the invocation. If H is a history, $complete(H)$ is the maximal subsequence of H consisting only of invocation and matching responses. 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 $res(e_1)$. Operations of different processes may be interleaved.

A history H is *sequential* if (1) the first event of H is an invocation, and (2) each invocation, except possibly the last, is immediately followed by a matching response. In other words, except for possibly the last event, a sequential history is a sequence of operations, i.e., pairs of invocation and matching response events. A *process subhistory*, $H|P$ (H at P), of a history H is the subsequence of all events in H whose process names are P. An *object subhistory*, $H|x$, is similarly defined for an object x. Two histories H and H' are *equivalent* if for every process P, $H|P = H'|P$. A history H is *well-formed* if each process subhistory $H|P$ of H is sequential.

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

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

Informally, $<_H$ captures the "real-time" precedence ordering of operations in H. Operations unrelated by $<_H$ are said to be *concurrent*. If H is sequential, $<_H$ is a total order.

A set S of histories is *prefix-closed* if, whenever H is in S, every prefix of H is also in S. A *single-object* history is one in which all events are associated with the same object. A *sequential specification* for an object is a prefix-closed set of single-object histories for that object. A sequential history H is *legal* if each object subhistory $H|x$ belongs to the sequential specification for x. Many conventional techniques exist for defining sequential specifications. We use the axiomatic style of Larch [11] and defer its description till after we define linearizability.

2.2. Definition of Linearizability

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

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

$L2$: $<_H \subseteq <_S$.

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

2.3. Specification Language

We use the Larch Specification Language [11] to specify the sequential behavior of an object. We use a Larch *trait* to specify its set of values and Larch *interfaces* to specify its set of operations. In a trait, the set of operators and their signatures, shown following the keyword **introduces**, defines a vocabulary of terms to denote values. For example, from the FifoQ trait of Fig. 1, *emp* and *ins*(*emp*,5) denote two different queue values. The set of equational axioms following the **asserts** clause defines a meaning for the terms, more precisely, an equivalence relation on the terms, and hence on the values they denote. For example, from FifoQ, we can prove that *rest*(*ins*(*ins*(*ins*(*emp*,3),4),3)) = *ins*(*ins*(*emp*,4),3). The **generated by** clause of FifoQ asserts that *emp* and *ins* are sufficient operators to generate all values of queues.

```
FifoQ: trait
  introduces
    emp: → Q
    ins: Q, E → Q
    first: Q → E
    rest: Q → Q
    isEmp: Q → Bool
  asserts
    Q generated by (emp, ins)
    for all (b: B, e, e1: 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


void Enq(queue q, elt e) {
    modifies q
    ensures q' = ins(q, e)
    }


elt Deq(queue q) {
    requires ¬ isEmp(q)
    modifies q
    ensures q' = rest(q) ∧ result = first(q)
    }
```

FIG. 1. FIFO queue trait and interfaces.

We use Larch/C *interfaces* [10] to describe an object's set of operations since all our implementations are in C. For example, interfaces for the *Enq* and *Deq* operations for FIFO queues are shown in Fig. 1. For an operation *op* of the object x of type T, an interface's header, like a C function header, is of the form *RT op*(T x, *args**) where *RT* is the type of the returned value of *op* and *args** is the list *op*'s arguments in addition to x. A **requires** clause states the precondition that must hold when an operation is invoked. An omitted **requires** clause is interpreted as equivalent to "**requires** true." A **modifies** clause lists those objects whose values are allowed to change as a side effect of executing the operation. An omitted **modifies** clause means no objects may change in value. An **ensures** clause states the postcondition that the operation must establish upon termination. An unprimed argument formal, e.g., q, in a predicate stands for the value of the object when the operation begins. A primed argument formal, e.g., q', stands for the value of the object at the end of the operation. The special reserved word *result*

denotes the value returned when *op* completes. We use the vocabulary of traits to write the assertions in the pre- and postconditions of an object's operations; we use the meaning of equality to reason about its values. Hence, the meanings of *ins* and = in the Enq interface are given by the FifoQ trait.

2.4. Two Simple Examples

The history

q Enq(5) A
q Enq(7) B
q Ok() A
q Ok() B
q Deq() C
q Ok(7) C

is linearizable because it is equivalent to the sequential history in which B enqueues 7 before A enqueues 5; hence C correctly dequeues the first element (7). The history

q Enq(5) A
q Ok() A
q Enq(7) B
q Ok() B
q Deq() C
q Ok(7) C

is not linearizable because the A enqueues 5 before B enqueues 7, but 7 is dequeued before 5. In these two examples, we rely on the FIFO semantics of queues. If q were a set and Enq and Deq had the standard semantics of "insert" and "remove" operations on sets, then the second history would be linearizable since insertion order on sets does not matter.

2.5. Implementation Issues: Definition of Correctness, Implementation Language

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

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

In this paper we implement each object's abstract operation in terms of atomic *instructions*, i.e., representation operations that are indivisible. These are the only representation operations of interest in our proofs of correctness. We use all uppercase letters for names of atomic instructions, e.g., FETCH_AND_ADD. The execution of each abstract-level operation corresponds to the sequential execution of these representation-level atomic instructions. For example, we might implement the enqueue operation at the abstract level as the sequence atomic instructions, FETCH_AND_ADD and READ, at the representation level. This implementation technique allows us at the representation-level to avoid using mutual exclusion locks, thereby permitting more concurrency at the abstract level.

All code for the simulation environment and our library of objects is written in C using a C-Threads package [5], which provides primitives to fork new processes and implement (at a lower level) atomic instructions like FETCH_AND_ADD. We used C primarily because we could use the C-Threads package to test our code on multiprocessors in our local computing environment. Though we could present the algorithms in a more perspicuous, abstract language, we rather present the actual working code. Moreover, we can present their formal specifications using the Larch/C interface language.

3. STRUCTURE AND USE OF THE SIMULATION ENVIRONMENT

3.1. The Simulation Package

Figure 2 shows the logical structure of the simulation package, where we use ovals to represent data and rectangles for procedures. Solid lines show control flow; dotted lines show data flow. The simulation package consists of several C functions stored in separate files. It uses C-Thread's *fork* primitive to create a user-specifiable number of threads to simulate a MIMD system.

There are three basic modules: *simulate*, *test*, and *analyze*. *Simulate* is the user's interface to the simulator; its main function is, in response to the user's request, to test whether a given implementation, *ConcObj*, exhibits only linearizable behavior with respect to a given specification, *SeqObj*. The user can specify various test conditions for the simulation, e.g., the number, N, of processes to run, how long (in number of operations or time) to run each process, and whether to use an input file of test cases (in the form of histories of events) or to generate a random set of test cases. Each test case corresponds to an object history of finite length. *Test* creates N processes, and invokes concurrently on their behalf a finite number of operations on *ConcObj*. After all N processes terminate, the simulator stores the resulting finite concurrent history in an event list, *History*, and then calls the *analyze* function to determine the linearizability of *History*. If an input file of (a finite number of) test cases is

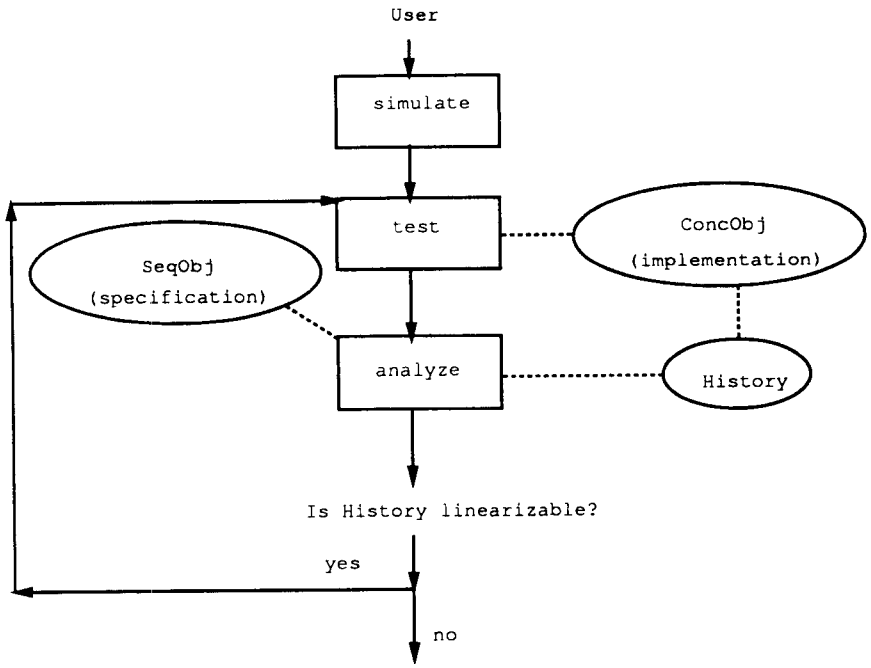

FIG. 2. Simulation package.

not given, *test* loops, thereby generating, upon the user's request, either an infinite or a finite number of finite histories to test on a given implementation; it stops if a history is found not to be linearizable. We have run the simulator on both single and multiprocessor architectures.

3.2. Example Uses of the Simulator

3.2.1. Simulating a Correct Implementation

Let us start with a concrete example, that of a FIFO queue whose specification is shown in Fig. 1. Below is an implementation where we store the elements of the queue in a list with head and tail pointers, *head* and *tail*, and keep track of the next free slot in the list in *back* (initialized to 1).

```
typedef struct elts {
    elt item;
        /* an element in the queue */
    struct elts *next;
        /* pointer to the next element */
    } *slotpt;

typedef slotpt slot;

typedef struct {
    slot head, tail;
        /* first and last slots in queue */
    int back;
        /* number of slots in queue */
    } reptype;
        /* queue representation*/

reptype queue;
```

Our FIFO queue implementation makes use of the following atomic instructions, whose behavior we specify below.

```
slot FETCH_AND_ADD (reptype Q, slot s) {
    modifies Q.tail, Q.back
    ensures result = s ∧ fresh(*(Q.tail'))
                       ∧ (*(Q.tail')).item = NULL
                       ∧ Q.tail' = (*(Q.tail)).next'
                       ∧ Q.back' = Q.back + 1
    }
```

FETCH_AND_ADD returns the slot s. It allocates a new, empty slot (with item = NULL), appends it to the list of elements in the original queue, and increases *Q.back* by 1. In Larch/C *fresh*(x) indicates that new storage, not aliased to any objects in the calling state, is allocated for object x.

```
void STORE(slot s, elt x) {
    modifies *s
    ensures (*s)'.item = x
    }
```

STORE puts an element x in a slot s.

```
int READ(int *x) {
    ensures result = (*x)
    }
```

READ returns the integer value pointed to by x.

```
elt SWAP(slot s, elt x) {
   modifies *s
   ensures result = (*s).item ∧ (*s)'.item = x
   }
```

SWAP puts an element x in the slot s and returns the old value of the item stored in s.

Given these atomic instructions, we implement the Enq and Deq operations for the FIFO queue as follows:

```
void Enq(queue q, elt x) {
    slot current;
    current = FETCH_AND_ADD(&q, q.tail);
                        /* get an empty slot */
        STORE(current, x); /* store x in slot */
    }
```

An *Enq* execution occurs in two distinct steps: a slot is atomically allocated (*back* is also increased) and the new element is stored as the *item* of the allocated slot.

```
   elt Deq(queue q) {
       int i, range;
       elt ch;
       slot current;

       while (true) { /* keep trying till
                         non-NULL value is found */
           current = q.head;
               /* starting from the first slot */
           range = READ(&(q.back)) - 1;
                 /* search up to back-1 slots*/
4      for (i = 1; i <= range; i++) {
           if ( i > 1) {
               current = current->next;
               }
           ch = SWAP(current, NULL);
              /* put a NULL value in ith slot */
           if (ch != NULL) {
                         /* if non-NULL value */
                      return(ch); /* return it */
               }
           }
       }
   }
```

Deq traverses the list of slots, starting at the first slot. For each slot, it atomically swaps NULL with the current *item*. If the value returned is not equal to NULL, *Deq* returns that value, otherwise it tries the next slot. If it has searched *back*-1 slots without encountering a non-NULL item, the operation tries again. *Deq* does not return until a non-NULL item is found.

To show a simple use of the simulator, here is a sample script of a simulation for the above FIFO queue. Our added comments to the script below are prefixed by *. Format of simulator output is the same as for input and matches our notation for histories given in Section 2. We use the notation $[x_1, \ldots, x_m]\{y_1, \ldots, y_n\}$ to denote the queue that contains in order the elements, $x_1, \ldots, x_m$, which are then followed in any order by zero or more of the elements, $y_1, \ldots, y_n$.

```
% simulate -cqueue-unbound -squeue -ooutput ⟨CR⟩
... set up simulation conditions here ...

/* This is simulation number 1 */
* initially, the queue is empty: Q = [] {}
Q   Enq(e)   P2     * P1, P2, P3 begin enqueueing.
Q   Enq(g)   P1
Q   Enq(a)   P3     * Q = [] {e, g, a}
Q   Ok()     P2
Q   Ok()     P1
Q   Deq()    P1
Q   Ok(e)    P1     * P1 removes first element, e,
Q   Enq(e)   P1     * and starts enqueueing e.
Q   Ok()     P3     * P3 finishes its enqueue of a,
Q   Deq()    P3     * starts to dequeue,
Q   Ok(g)    P3     * and removes g.
Q   Ok()     P1     * Q = [] {e, a}, i.e.,
                    * Q = [e, a] or Q = [a, e]
/* This history is linearizable! */
```

Here we see that three processes, $P1$, $P2$, and $P3$ execute concurrently on queue object Q. The three processes all start to enqueue items with $P2$'s enqueue of e finishing first and then $P1$'s enqueue of g. When $P1$ then dequeues an item it sees what $P2$ enqueued, thus inducing an ordering of e before g in Q. $P1$ then starts to enqueue another e. $P3$ finally finishes its enqueue of a and then dequeues the g that $P1$ had enqueued. Finally, $P1$'s enqueue of e finishes, leaving Q containing both e and a, but from the outside observer's viewpoint in either order since those two items were enqueued concurrently.

3.2.2. *Detecting that an Implementation is Incorrect*

We intentionally inject an error in the implementation of the queue example and then simulate it. We modify the *Deq* operation so that a process could possibly dequeue an element that was inserted after the current "first" element in the queue.

```
elt Deq(queue q) {
    int  i, range;
    elt ch;
    slot current;

    while (true) {
        current = q.head;
        i = 1;
        range = READ(&(q.back)) - 1;
        while (i <= range) {
                     /* search up to back-1 slots*/
            if (i > 1)
                current = current->next;
```

```
            ch = SWAP(current, NULL);
            if (ch != NULL) {
                return(ch);
                }
            i++;
            range = READ(&(q.back)) - 1;
            /*** modify current search range     ***/
            /*** queue.back could have been      ***/
            /*** changed by some other process   ***/
            }
        }
    }
```

Running this version of the FIFO queue through our simulator yielded the following nonlinearizable history:

```
/* This is simulation number 10*/    * initially,
  the queue is empty
Q   Deq()   P1
Q   Deq()   P2
Q   Deq()   P3
Q   Enq(f)  P4
Q   Ok()    P4          * Q = [f] {}
Q   Enq(y)  P4
Q   Ok(f)   P1          * Q = [] {y}
Q   Ok()    P4          * Q = [y] {}
Q   Enq(t)  P4
Q   Enq(e)  P1
Q   Ok()    P4          * Q = [y] {t, e}
Q   Enq(o)  P4
Q   Ok()    P1          * Q = [y] {t, e, o},
Q   Deq()   P1          * though t must precede o.
Q   Ok(e)   P2          * Wrong! e cannot be the first
Q   Enq(c)  P2          * element of Q.
Q   Ok(y)   P1
Q   Ok()    P2
Q   Ok(c)   P3
Q   Ok()    P4

/* This history is not linearizable! */
```

$P2$ should not be able to dequeue e since y is the first element in the queue. The first element is y because the response event of $P4$'s enqueue of y precedes the invocation event of $P1$'s enqueue of e.

Considering the details of the implementation, the order in which the (relevant) atomic instructions occur is

- $P4$ does a FETCH_AND_ADD, reserving a slot for y;
- $P2$ does a READ, setting its range; when it does a SWAP, scanning the slot for y it finds a NULL value;
- $P4$ does its STORE of y;
- $P1$ reserves a slot for e and stores e in it (FETCH_AND_ADD and STORE);
- $P2$ increases its search range by re-READing $Q.back$ (which has been changed) and continues scanning to the next slot, that containing e;
- $P2$ gets e.

In short, $P4$'s STORE occurs after $P2$ scans the slot reserved for y. Since subsequent pending, concurrent enqueues modify $q.back$, $P2$'s search range increases, so $P2$ continues searching, encountering e, which $P1$ enqueues before $P2$'s next SWAP. In the correct version, $P2$ would instead exit that iteration of the search loop (because range would be fixed) and start a new scan of the queue from the first slot. We argue in Section 6.3.1 in general why the modified Deq operation is incorrect.

4. THE LINEARIZABILITY ANALYSIS ALGORITHM

For a given data type T, to effectively test linearizability of a concurrent implementation, *ConcObj*, against a sequential specification, *SeqObj*, we use a sequential implementation of T as a specification. In this sense, our sequential implementation serves as an "operational" or "executable" specification. We assume that the sequential implementation (e.g., expressed in C) satisfies the object's sequential specification (e.g., expressed in Larch).

The essence of the analysis algorithm is as follows: given a history H of a concurrent object, *ConcObj*, we try every possible sequential order of H's concurrent operations while preserving its real-time order relation $<_H$. We check if each sequential history H_S is linearizable by executing the operations on *SeqObj*. If every possible ordering of H fails, by the definition of linearizability, the history H is nonlinearizable.

4.1. Data Structures and Algorithm

A history H is stored in a doubly linked list of *events*,

```
typedef struct ev {
    char item;
    char op;
    char name[10];
    struct ev *match, *next, *prev;
    } event;
```

where *item* is the argument for the operation *op* and *name* is the name of the process that invoked *op*. *Next* and *prev* are two pointers pointing to the previous and next *events* in H, respectively. For an invocation event, *match* points to its matching response event. We use the special *NULL* event as a sentinel and put it at the end of the history. Figure 3 depicts a snapshot of a history represented as a list of events.

DEFINITION. A *section* of a history H is an invocation event, its matching response event, and all events in between them.

Sections of a history H can be ordered by the positions of their first events in H. Figure 3 shows three sections in the history H.

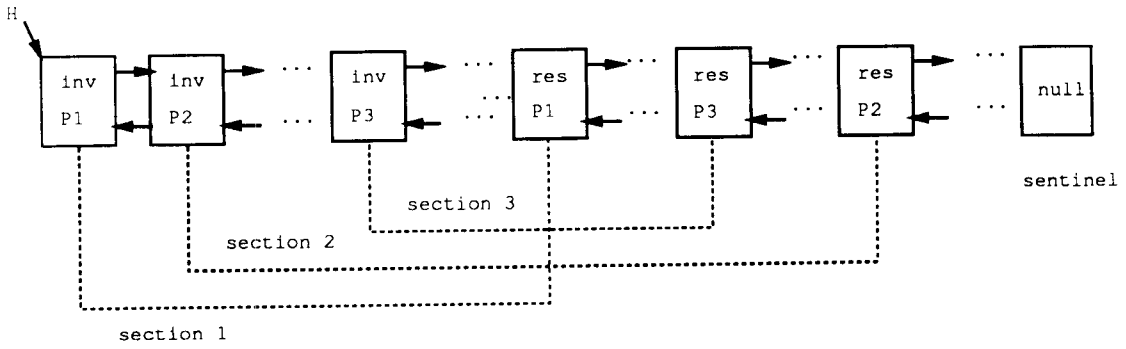

FIG. 3. Snapshot of a history.

Analyze (recall from Fig. 2) calls the procedure *search* to search iteratively over events.

```
bool analyze(event *history) {
    bool linearizable;
    event *p;

    linearizable = ((p = search(history))
      != NULL);
    return(linearizable);
    }
```

The heart of the simulator is the *search* procedure. If a history H is linearizable, *search* returns a linearization of H; otherwise it returns with an empty list of events. The *search* function uses a stack to keep track of the portion of H that is linearizable so far. Conceptually, the stack elements are operations (both invocation and response events per operation); actually, the stack elements are pointers to the operations' events in H (see Fig. 4).

```
typedef struct {
    event *pi, *pr, *inv, *resp;
    char item, op, result;
    } elt_stack;
```

```
typedef struct {
    elt_stack value[STACK_LENGTH];
    int in;
    } stacktype;
```

We use *pi* and *pr* to locate the first *section* of the subhistory that has not yet been checked; *inv* and *resp* to locate the operation that we select as the first operation of the subhistory; *item*, *op*, *result* to remember this operation.

We implemented the stack procedures, *push*, *pop*, *top*, and *isempty*, with their usual semantics, and also the following auxiliary procedures on event lists: *p_copy* makes a local copy of an operation; *lift* temporarily "removes" the invocation and response events of an operation from the history (Fig. 5); *unlift* puts the invocation and response events back in their original positions in the history-needed if we later backtrack in our search procedure; *linearization* links the operations stored in the stack, creating and returning a linearization of the history.

Informally, *search* always looks for the next operation in H that could happen (but not conflict with the partial

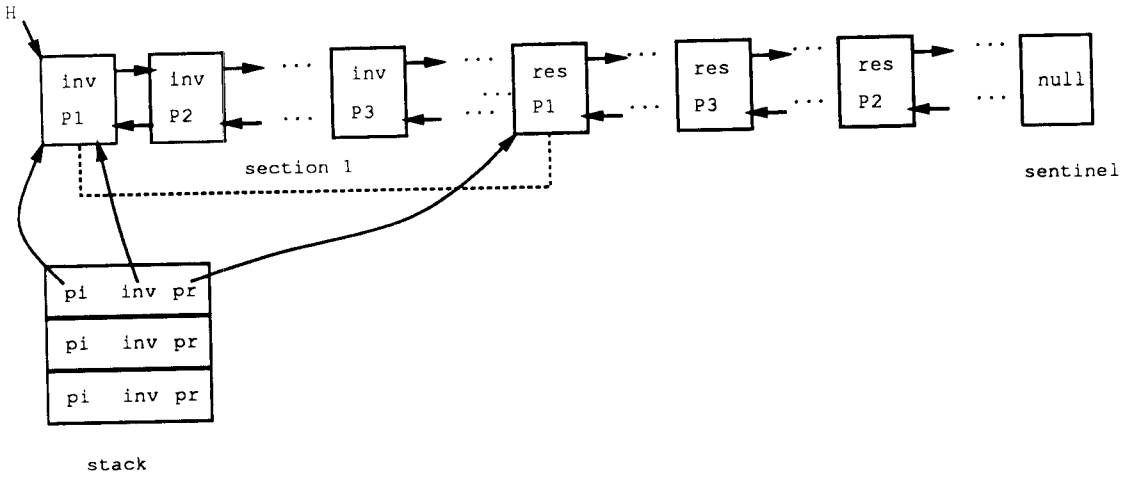

FIG. 4. Top of the stack tracks current section and operation.

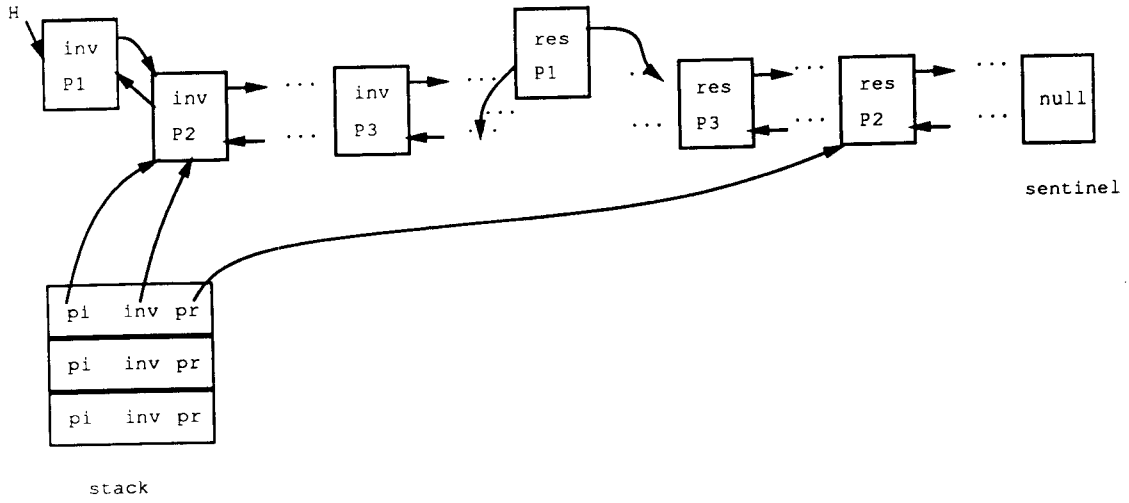

FIG. 5. Lifting an operation in a history.

order relation $<_H$) from the current *section* and tries to form a legal history. Once it decides that an operation could happen, it conceptually pushes the operation onto the stack and lifts the events of this operation from the history; it repeats this procedure on the remaining history until the remaining history is empty. During this procedure of rearranging the operations of a history, we might need to undo some operations on *SeqObj* if we find that a tentative reordering of a subhistory of H is nonlinearizable.

More specifically, we sketch below the gist of the *search* procedure, given in the Appendix in its entirety:

1. Initialize the stack.
2. Locate the current *section* of H by the *pi* and *pr* pointers of *current* if there is one (see Fig. 4); otherwise return a pointer to the linearized history.
3. From the current section, select an operation and store it locally in *current*.
4. Simulate this selected operation on the sequential implementation of the object by calling *operation*, corresponding to the selection operation.
5. (a) If *operation* returns *true*, meaning those events checked so far consists of a linearizable subhistory, then push this operation onto the stack, lift this operation from the history H and go back to 2.

(b) Otherwise:

(i) If in the current section there is still some unselected operation whose invocation event is not preceded by any response event, then select one and go back to 4.

(ii) At this point, every operation in the current section has been tried without success. So we have to backtrack to the previous section to try another arrangement. If the *stack* is empty, meaning that the history is not linearizable, then return a *NULL* value. Otherwise, (1) get the top element of the stack, which contains all information about the previous section and selected operation, by an auxiliary pointer (called *tmp* in the code of the Appendix), and pop the stack; (2) undo the previous operation and put it back to the history (see Fig. 6); (3) set *current's* pointers to the previous section and operation; and then finally (4) go to 5(b)i.

If the history is linearizable, then this procedure returns a pointer to the first event of the history, otherwise it returns a *NULL* value.

4.2. The Correctness of the Search Algorithm

We give an informal correctness argument for the *search* algorithm. For a given history H, and a linearizable subhistory H_1 of H, we maintain the following invariants:

I_1: If $top = i$, then the operations pointed by the pointers *inv* and *resp* in *stack*[1], *stack*[2], ..., *stack*[$i - 1$] form a linearizable subhistory H_1 of H and $<_{H_1} \subseteq <_H$.

I_2: The pointers *pi* and *pr* in *stack*[*top*] identify the first *section* of the subhistory $H - H_1$ (H with those operations of H_1 removed), and *inv* and *resp* in *stack*[*top*] identify an operation whose invocation event is in the first *section*.

I_3: The auxiliary pointer *tmp* always points to the head of the first *section* in $H - H_1$.

We claim that *search* returns a non-NULL value if and only if H is linearizable, and argue inductively as follows:

1. *Search* can return a non-NULL value only at Step 2 and only under the condition that the remaining history is empty. At this point, we have processed the entire his-

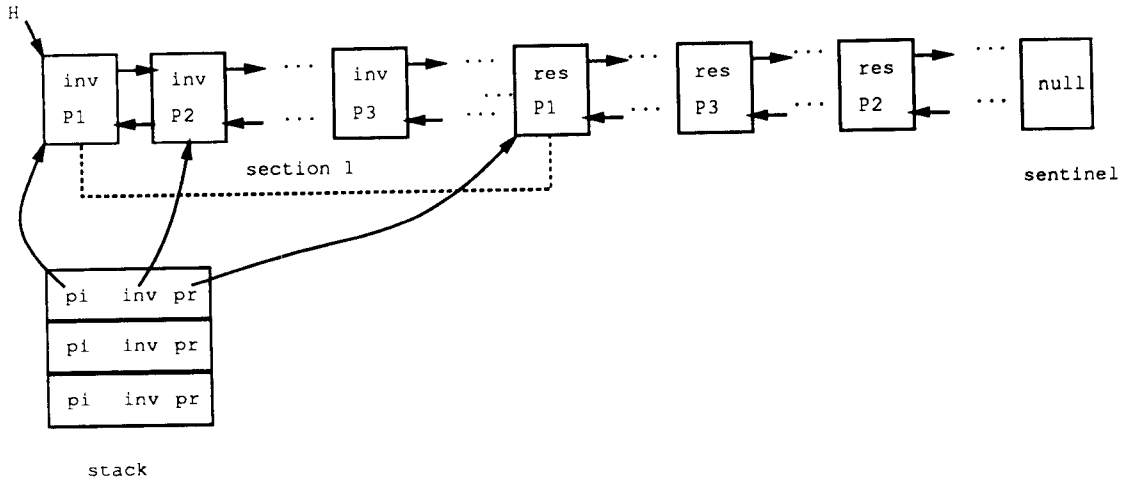

FIG. 6. Select another operation.

tory successfully, identifying a linearization of H operations stored from $stack[1], \ldots, stack[top]$.

2. Suppose the history H is linearizable, then there must be a sequential H' such that H' is legal and $<_H \subseteq <_{H'}$ (definition). According to the definition of the relation $<_H$, the first operation of H' must be in the first section of H and cannot be preceded by any response event (Step 5(b)i). *Search* will eventually select this operation as the first operation of the remaining subhistory $H - H_1$, remove it from H (Step 5), and check the remaining history H_1. If H_1' denotes the history H' with its first operation removed, then we have $<_{H_1} \subseteq <_{H_1'}$. Using similar reasoning, we can prove that *search* will arrange H in the same order as in H'. Since H' is legal, i.e., it is linearizable, *search* will return a non-NULL value.

4.3. Runtime Performance

There exists simple data types and histories for which testing linearizability is NP-complete. Thus, since we intentionally built the simulation environment to work in general, i.e., for any data type, our analysis algorithm is exponential in time. This means that analyzing a long history (say, 1 million operations) is impractical; however, analyzing many short (100 operations) histories is tractable. Thus, a typical way to use our simulator is to generate and analyze short histories over a long period of time, e.g., days or even months (if desired). Space is not a limiting factor since the simulator discards each history it determines is linearizable.

For a concrete idea of how the simulator performs in practice, testing a history of 60 FIFO queue operations takes the simulator about 2 s; a history of 100 operations, about 1 min. For a concurrent set example, it is even faster; a history of 200 operations takes 2 min.

For detecting an incorrect history, we also have a better chance of finding a nonlinearizable history by testing many short histories rather than testing one long one. For the queue example, we generated and tested about 100 histories, each of 50 operations, before we encountered a nonlinearizable history; for the set example, we generated about 70 histories, each of 60 operations.

5. VERIFYING CONCURRENT OBJECTS

5.1. Library of Concurrent Objects

We have implemented, simulated, and verified concurrent versions of the following data objects:

- unbounded FIFO queues,
- bounded FIFO queues,
- unbounded priority queues,
- bounded priority queues,
- semiqueues,
- stuttering queues,
- sets,
- "multiple" sets,
- read/write registers, and
- B-trees.

In the next three sections we present the specifications, implementations, and proofs of correctness for FIFO queues (unbounded and bounded versions), semiqueues, and stuttering queues. Why queues? Queues are prevalent in most operating systems, programming languages, and application software. They are needed for task scheduling, resource management, message buffering, and event handling. They can be the cause of performance bottlenecks since multiple processes need to access them. An implementation using more tradi-

tional approaches, e.g., mutex locks, would block out all other processes if some enqueueing or dequeueing process currently has access to the shared queue. Using linearizability as the correctness condition permits enqueuers to proceed concurrently, and enqueuers and dequeuers to proceed concurrently when the queue is nonempty. Hence, we permit higher degrees of concurrency than normally provided by a more traditional implementation.

We also choose to focus here on these three types of queue-like structures since related work covers the relevant aspects of the other kinds of objects. In particular, the designs and proofs of correctness of the implementation for the semiqueue and stuttering queue are new contributions of this paper. The designs and proofs of correctness of the implementations of bounded and unbounded versions of linearizable priority queues are new too, but similar enough to those of the bounded and unbounded versions of the FIFO queue not to warrant a detailed presentation.

Finally, others have designed implementations of the nonqueue data types listed above: Lamport's atomic registers [18], Lanin and Shasha's sets [20], Lehman and Yao's B-trees [21], and Weihl and Wang's extensions to B-trees for multiversion memory [25]. These implementations all satisfy the linearizability property. Their proofs of correctness can be found in the original references.

5.2. Remarks on Our Proofs and Proof Method

Consider again the FIFO queue. Its sequential specification as expressed in Larch makes sense only when there is a total order relation between the items in queue, so as to know which is the *first* item in the queue. If we perform *Enq* and *Deq* operations sequentially, we get a natural total order relation on the queue's items. Suppose now we do multiple *Enq* operations concurrently. What ordering relation can we define that will still give meaning to *first*?

The answer in general is that the implementor of a concurrent object needs to define an ordering relation. This ordering (typically a partial order) is defined in terms of the representation operations (typically the atomic instructions) used to implement the abstract operations of the concurrent object. A proof of correctness amounts to showing that the implementation maintains the consistency of this ordering.

As mentioned in the introduction, proving the correctness of an implementation can be a daunting task. In this paper we follow no formal proof method in the sense of writing a syntax-directed proof or a machine-checkable proof. Rather, we focus on providing the key insight, the ordering information, needed for performing any proof of correctness. In other words, we are verifying the correctness of the algorithms, not the program text that expresses them.[1] In the following examples, the reader should find evident reusable patterns in the proofs' structure.

Linearizability is only a safety property. We make no claims about our implementations satisfying any liveness properties, e.g., that they are *wait-free* [12]. Herlihy proposes a method for implementing *non-blocking* and *wait-free* concurrent objects [13] and thus addresses liveness directly. In order to guarantee progress is made, we assume a nonmalicious and fair scheduler as part of the environment of our concurrent systems.

6. FIFO QUEUES

6.1. Specification

Given the specification for FIFO queues (Fig. 1), we provide two different implementations, one using an unbounded amount of storage and another using a bounded amount. We provide the unbounded version first because its implementation and corresponding proof of correctness are simpler to understand. The implementation and proof also both provide a reusable general framework for the implementations and proofs for the semiqueue and stuttering queue examples.

6.2. Implementation for the Unbounded Version

The implementation is given in the beginning of Section 3.2.1.

6.3. Proof of Correctness for the Unbounded Version

Given a well-formed history H, we use the following notation:

- H_i is the ith event in H. We assume that an invocation event of an abstract operation is associated with the execution of the first instruction of the operation and a response event associated with the execution of the last instruction.
- If $H_i = e$ then $label(e) = i$.
- $|H|$ is the length of H.
- $[H_i]$ is the subhistory of H consisting of the first i events in H.
- If the operation Enq(x)/Ok() is in H, we use $enq(x)$ and $eok(x)$ to refer to its invocation and response event; if the operation Deq()/Ok(x) is in H, we use $deq(x)$ and $dok(x)$ to refer to its invocation and response event. For simplicity, but without loss of generality, we assume that no two items are equal.

[1] A formal proof of correctness of C code would be difficult to do without a formal semantics of C.

• $enqueue(H) = \{x \mid$ Both $enq(x)$ and $eok(x)$ are in $H\}$. Note this is a partially ordered set with the relation $<_r$ (see below) and we define the concept of *chain* as usual.
• $dequeue(H) = \{y \mid$ Both $eok(y)$ and $deq(y)$ are in H, and if $dok(y)$ is in H, then $label(eok(y)) < label(dok(y))\}$.
• $left(H) = enqueue(H) - dequeue(H)$.

We define a partial order relation $<_r$ on the items in the list

$x <_r y$ iff the STORE for x precedes
the FETCH_AND_ADD for y.

We will be interested in any total order relation consistent with $<_r$.

We observe that

1. Several *Enq*'s operations can occur concurrently.
2. If the *queue* is not empty, a *Deq* can execute with several *Enq*s concurrently and several *Deq*s can also occur concurrently. Only one *Deq* can get the first item.
3. In *Deq* we limit the search *range* in terms of the current value of *queue.back* and then search from 1 to *range*. While one process is searching within the range, some other process might do an *Enq*, increasing the value of *queue.back*. For example, if we replace *range* in line 4 (see code in Section 3.2.1) by *queue.back*, we would get incorrect behavior since doing so would allow a process to delete an item that would not be the first one by any total order relation defined above.

LEMMA 1.1. *An item in the queue can be deleted at most once.*

Proof. The only way to remove an item from *queue* is to execute the atomic instruction SWAP. Since SWAP is atomic and places NULL in the slot no other process may delete the item. ■

LEMMA 1.2. *If H is a history accepted by the unbounded FIFO queue, then for all i, $1 \le i \le |H|$, $dequeue([H_i]) \subseteq enqueue([H_i])$.*

Proof. If not, there must be an integer $i > 0$ and an item x such that $label(dok(x)) = i$ and $label(eok(x)) = j > i$, which means that x is deleted before it had been stored, a contradiction. ■

LEMMA 1.3. *If H is a history accepted by the unbounded FIFO queue, $H_i = enq(x)$ and $H_j = eok(x)$ with $i < j$, then x is a maximal element in $enqueue([H_j])$.*

Proof. Suppose not. Then there must be a $y \in enqueue([H_j])$ and $x <_r y$. According to the definition of $<_r$, the FETCH_AND_ADD instruction for y must be executed after the execution of the STORE instruction for x, which implies that $label(eok(x)) = j < label(enq(y))$. So, $enq(y)$ cannot be an event in $[H_j]$, let alone y be in $enqueue([H_j])$. ■

LEMMA 1.4. *Suppose H is a history accepted by the unbounded FIFO queue, $eok(x)$ and $eok(y)$ are in H. If x and y are stored in the mth slots and the nth slots, respectively, then $label(enq(x)) < label(enq(y)) \Leftrightarrow m < n$.*

Proof. By the semantics of FETCH_AND_ADD and Q.back monotonically increases. ■

LEMMA 1.5. *When a process P swaps out an item x from the mth slot (by executing SWAP), x must be the minimal element with respect to the relation $<_r$ in the set $S = \{x \mid x$ is stored in some slot between the first and mth slots, inclusive.$\}$*

Proof. Observe that once the value of *range* is changed, P searches the list from the first slot and $m <$ *range*. Suppose P swaps x from the mth slot, then the value of *range* must be the same as when P starts searching from the first slot, so for any item y stored between the first slot and the slot $m - 1$, $label(enq(y)) < label(deq(x))$. For any two items, x_a and x_b, stored in the lth and nth slots, where $1 < l < n < m$, it is impossible that $x_b <_r x_a$ by Lemma 1.4. If $x_a <_r x_b$, then $label(eok(x_a)) < label(enq(x_b)) < label(deq(x))$, i.e., when P searches the lth slot, x_a must have been already stored there; hence P would get the item x_a (if it is still there). ■

THEOREM 1. *If H is a complete history accepted by the unbounded FIFO queue implementation, then H is linearizable.*

Proof. We prove the linearizability of H by induction on the number of operations in H. For H with 0 operations, it is trivial that Theorem 1 holds. Assume that for any complete H with l operations the theorem is true. We need to show that for any complete H with $l + 1$ operations it is also true. Note that $|H| = 2(l + 1)$. Since H is a complete history, the last event of H must be a response event. There are two possibilities:

(1) The last event is $H_{2(l+1)} = eok()$ and its matching invocation event is $H_j = enq(x)$ for some x and $1 \le j < 2(l + 1)$. We use H_{old} to denote H with the two matching events H_j and $H_{2(l+1)}$ deleted. Since x cannot be dequeued in H_{old}, H_{old} is also a complete history accepted by the unbounded FIFO queue and so it is linearizable, equivalent to a legal sequential history H'. By Lemma 1.3, $H'H_jH_{2(l+1)}$ is a legal sequential history and equivalent to H. So H is linearizable.

(2) The last event is $H_{2(l+1)} = dok(x)$ for some x and its matching invocation event is $H_j = deq()$, $1 \le j < 2(l + 1)$. The same arguments as above applies with Lemma 1.3 replaced by Lemma 1.5. ■

6.3.1. Revisiting the Incorrect Implementation

Our proof of correctness relies on the *Enq* and *Deq* operations preserving the following two properties of our partial order $<_r$:

1. An *Enq* adds an item x that is maximal with respect to $<_r$.
2. A *Deq* removes and returns an item x that is minimal with respect to $<_r$.

Our incorrect implementation of the queue given in Section 3.2.2 still preserves the first property (enqueueing a maximal element), but not the second. A process executing a *Deq* operation could dequeue an element that is not minimal with respect to $<_r$. Suppose process A is in the middle of performing an *Enq*(x) on an empty queue and just finished FETCH_AND_ADD (*back* = 2). Now process B starts a *Deq*, finding nothing in its first iteration (since A has not finished its *STORE*). It is possible that before B rereads *back*, A finishes its STORE (x is in position 1) and then another process C also finishes an *Enq*(y) (*back* = 3 and y is in position 2). If at this point B rereads *back* and enters the loop the second time, what is going to happen? B will remove and return y instead of x. (Recall that $x <_r y$ according to our definition of $<_r$.)

6.4. Bounded Version

In order to use a fixed amount of storage, in our second implementation we use a bounded array to store the elements of the queue. We use modular arithmetic to "fold" an unbounded list into the bounded array. To preserve the FIFO ordering, each queue element is tagged with a generation number that counts the number of times the *back* counter has "wrapped around."

```
typedef struct {
    elt item;           /* a queue element */
    int tag;            /* its generation number */
    } entry;

typedef struct rep {
    entry elts[SIZE];   /* a bounded array */
    int back;
    } reptype;

reptype queue;
```

In addition to the READ instruction described in the unbounded version, the bounded version makes use of the following atomic instructions:

```
entry EXCHANGE(entry e1, int gen, entry e2){
  modifies e1
  ensures result = e1 ∧ (e1.tag = gen ⇒ e1' = e2) ∧
    (e1.tag ≠ gen ⇒ unchanged(e1))
  }
```

EXCHANGE returns the old value of e1. If e1.tag matches gen, then it sets e1 to the value of e2; otherwise it leaves e1 unchanged.

```
void FETCH_AND_MAX(int *i, int j) {
  modifies *i
  ensures (*i)' = max((*i), j)
  }
```

FETCH_AND_MAX sets the integer pointed to by i to be the maximum of its initial value and j.

Enq is implemented as

```
void Enq(queue q, elt x) {
    int i;
    entry e, *olde;

    e.item = x;
          /* set the new element's item to x */
    i =  READ(&(q.back)) + 1;
          /* get a slot in the array for the new
          element */
    while (true) {
        e.tag = i / SIZE;
      /* set the new element's generation number */
          olde = EXCHANGE(&(q.elts[i % SIZE]),
                          -1, &e);
            /* exchange the new element with
     slot's value if that slot has not been used */
      if (olde→tag == -1) {  /* if exchange
                              is successful */
         break; /* get out of the loop */
         }
      ++i; /* otherwise, try the next slot */
      {
 FETCH_AND_MAX(&(q.back), i); /* reset the
                                  value of back */
}
```

Initially each entry's tag is equal to -1 and *back* = -1.[2] *Enq* reads the index of the last enqueued item, and cyclically scans the array starting at the slot after that index. EXCHANGE checks whether each slot is empty, and if so, swaps in the item x tagged with its generation number. If the tag of the entry returned is NULL, then the slot was empty and *queue.back* is updated to the maximum of i and the current value of *queue.back* (other concurrent *Enq*'s could have updated *queue.back* before this one completes).

```
elt Deq(queue q) {
    entry e, *olde;
    int i, range;

    e.tag = -1;          /* make e an empty entry */
    e.item = NULL;
    while (true) {
        /* keep trying until an element is found*/
      range = READ(&(q.back)) - 1;
                /* search up to back-1 slots */
      for (i = 0; i <= range; i++) {
          olde = EXCHANGE(&(q.elts[i % SIZE]),
                          i / SIZE, &e);
```

[2] C arrays start indexing from 0.

```
/* check slot to see if it contains the oldest
   element */
            if (olde->tag != -1) { /* if so */
                return(olde->item);
                        /* return the item in it */
                }
                    /* otherwise try the next one */
            }
        }
    }
```

Deq cyclically scans the array, starting at index 0 and ending at the observed value of *queue.back*. For each element, it atomically compares to see if its *tag* is the current generation number; if so, it swaps in the "empty" entry. If the *tag* of the entry returned is not equal to -1, then *Deq* returns the associated item.

6.5. Proof of Correctness for the Bounded Version

We change the definition of $<_r$ to be as follows:

$x <_r y$ iff FETCH_AND_MAX for x precedes the READ for y.

Lemma 1.1 still holds. Again we assume that only distinct items are inserted in a queue.

LEMMA 2.1. *Suppose H is a history accepted by the bounded FIFO queue and $H_j = eok(x)$, then x* is a maximal element in *enqueue(H_j).*

Proof. The same argument as in Lemma 1.3 with the FETCH_AND_ADD instruction replaced by READ, and STORE by FETCH_AND_MAX.

We need the following new notation:

- For an item x, *entry*(x) denotes the entry holding x.
- $i(x)$ for the index of *queue* where x (or *entry*(x)) is stored.
- $slot(x) = i(x) + (SIZE * entry(x).tag)$. ■

LEMMA 2.2. *If H is a history accepted by the bounded FIFO queue and $x <_r y \in enqueue(H)$,* then $slot(x) < slot(y)$.

Proof. We use *back* to remember the most recently used index of a slot. An enqueueing process P first gets a slot index by reading *back*+1 (several concurrent processes may get the same index); then it atomically does the following: (a) checks if the slot is empty, and (b) if so stores the item there, prohibiting other concurrent processes from using this slot again. After storing the item, P will increase the value of *back* by at least 1 through FETCH_AND_MAX. Since $x <_r y$, according to our definition of $<_r$, we know that the operation READ for y must be after the FETCH_AND_MAX for x. So y's enqueuer can get only a greater *slot* value. ■

LEMMA 2.3. *If H is a history accepted by the bounded FIFO queue and $H_j = dok(x)$, then x is the minimal item of left($[H_j]$).*

Proof. If not, there must be a $y \in left([H_j])$ such that $y <_r x$; by Lemma 2.2, $slot(y) < slot(x)$, since a process starts its search range from the first slot and its search range is limited by *back*. Hence, the same reasoning as in Lemma 1.5 applies here. ■

THEOREM 2. *If H is a complete history accepted by the bounded FIFO queue implementation, then H is linearizable.*

Proof. Similar to the proof of Theorem 1 using Lemmas 1.1 and 2.1–2.3. ■

7. SEMIQUEUES

7.1. Specification

A *Semiqueue$_k$* object consists of a sequence of items. The *Enq* operation inserts an item in the sequence, and the *Deq* deletes and returns one of the first k items in the queue. If k is one, the object is a FIFO queue (Fig. 1) and if k is n, the maximum number of items allowed in the queue, the object is a multiset. The motivation for providing this "weaker" queue data type [17] is to give better response time to dequeueing processes.

The Larch specification of the *Semiqueue$_k$* object shown in Fig. 7 shows a reuse trait: if a trait T **includes** another trait T_1, then T extends the theory denoted by T_1 by adding more operators and equations explicitly in T. The SemiQ trait includes the FifoQ trait of Fig. 1 and the Set trait (not shown) and extends them by adding two operators, *prefix* and *del*.

```
SemiQ: trait
  includes FifoQ, Set
  introduces
    prefix: Q, Int → SetOfE
    del: B, E → B
  asserts for all [q: Q, i: Int]
    prefix(q, i) = if (i = 0 ∨ isEmp(q)) then {} else prefix(rest(q), i-1) ∪ {first(q)}
    del(emp, e) = emp
    del(ins(b, e), e1) = if e = e1 then b else ins(del(b, e1), e)


void Enq(queue q, elt e) {
   modifies q
   ensures q' = ins(q, e)
   }


elt Deq(queue q) {
   requires ¬ isEmp(q)
   modifies q
   ensures q' = del(q, result) ∧ result ∈ prefix(q, k)
   }
```

FIG. 7. *Semiqueue$_k$*.

7.2. Implementation

Our implementation of a $Semiqueue_k$ uses the same list representation of queues as that for the unbounded version of the FIFO queue. The *Enq* operation is identical. To implement the *Deq* operation, we use the following atomic instruction:

```
void ADD(int *p) {
  modifies *p
  ensures (*p)' = (*p) + 1
  }
```

ADD atomically increases by one the value of the integer pointed to by p.

Deq is as follows:

```
    elt Deq(queue q) {
        int i, range, differ;
        elt value;
        slot current;

1   while (true) {
        current = q.head;
3       range = READ(&(q.back));
            /* range initially is back */
        i = 0;
        i = 0;
            /* starting search from first slot */
        while ((i < range) && (i < q.back)) {
            value = SWAP(current, NULL);
            if (value != NULL) {
                        /* successful dequeue */
                ADD(&num_deqd); /* number of
                            items actually deq'd */
                return(value);
                }
            i++; /* try the next location */
            current = current->next;
            differ = range - num_deqd;
                 /* differ is the number of items
                            potentially enq'd but */
                 /* not yet deq'd (and possibly not
                    yet stored) from 1 to range. */
13          if (differ < K) {
14              range += K - differ;
                    /* ok to incr range since there
                                                were */
                    /* fewer than K items from 1 to
                                                range */
                }
            }
        }
    }
```

The while loop in *Deq* is similar to that for the unbounded FIFO queue except that *range* can be increased during the search. The loop invariant is that *num_deqd* is always less than the number of items deleted. When a dequeueing process starts its first search, setting *range* = *back* establishes the invariant since any inserted items between the first slot and the *range*th slot are in order. After the first search, we can allow the dequeueing process to search further if we can ensure that it would not reach an item that is not in the first K items of the queue. The local variable, *differ*, keeps track of the number of items potentially enqueued that have not yet been dequeued within the slots 1 to *range*. (Recall that some enqueueing process might have reserved a slot between 1 and *range* but have not yet done a STORE of it.) By checking *differ* in the last test (lines 13 and 14), we check that there are at most *differ* items that could be ordered before those items in the slots after *range*, so we are safe in increasing *range* by K − *differ*.

7.3. Proof of Correctness for the Semiqueue

The partial order is the same as defined for the unbounded FIFO queue. Lemmas 1.1, 1.2, 1.3, and 1.4 still hold here (with unbounded FIFO queue replaced by semiqueue).

LEMMA 3.1. *num_deqd is always less than or equal to the number of items deleted from the queue.*

Proof. Initially, we set *num_deqd* = 0. *Num_deqd* is changed only at one place in *Deq* and only after a process has deleted an item from the queue. Once an item is deleted by some process by executing the SWAP and the item is non-NULL, this dequeueing process increases *num_deqd* by 1. ■

LEMMA 3.2. *When a process P swaps out an item x from the mth slot (by executing SWAP), the set (partially ordered by $<_\Gamma$) $S = \{x \mid x$ is stored in slot between the first slot and the mth slot$\}$ contains no chain with length $\geq K$.*

Proof. Observe that the value of *range* can be changed at only two places, at the beginning of the loop at line 3 (because of some concurrent enqueueing process) and at the end of the loop at line 14 (because of this dequeueing process). We define *Phase 1* to be the interval from when *range* is changed at the beginning of the loop to when it is changed at the end, and *Phase 2* to be the interval from when *range* is changed at the end of the loop to when it is changed at the beginning. During the execution of a *Deq*, a process passes through these two *Phases* alternatively.

By Lemma 3.1 and the test in line 13, we can prove the following properties about *range*:

(1) $m <$ *range*.

(2) In *Phase 2*, there can be at most K items between the first and *range*th slots.

So if P gets x in *Phase 2*, the Lemma must be true since there are less than K items between slots 1 and *range*.

Suppose P swaps x in *Phase 1*, then the value of *range* must be the same as when P starts searching from slot 1, so for any item y stored between slot 1 and $m - 1$, *label*(*enq*(y)) < *label*(*deq*(x)). For any two items, x_a and

x_b, stored in slots l and n, $1 < l < n < m$, it is impossible that $x_b <_r x_a$ by Lemma 1.4. If $x_a <_r x_b$, then $label(eok(x_a)) < label(enq(x_b)) < label(deq(x))$, i.e., when P searches slot l, x_a must have been stored there. Thus P can get x_a (if it is still there). In this case, we showed that there is no chain with length >1 on the set of elements in slots 1 to $m - 1$. ■

LEMMA 3.3. *If H is a history accepted by the semiqueue and x is an item such that $H_i = deq(x)$, $H_j = dok(x)$, then $x \in enqueue([H_{j-1}])$ and there is no chain C in left $([H_{j-1}])$ such that x is after the Kth item on C.*

Proof. Since $H_j = dok(x)$, we have $enqueue([H_{j-1}]) = enqueue([H_j])$. We also know that $x \in dequeue([H_j])$. By Lemma 1.2, $x \in enqueue([H_j])$, so $x \in enqueue([H_{j-1}])$.

Suppose that there is a chain C in $left([H_{j-1}])$,

$$x_1 <_r \ldots <_r x_k <_r \ldots <_r x <_r \ldots,$$

then $label(eok(x_l)) < label(enq(x)) < j$, $(1 \le l \le k)$ and no event $deq(x_l)$ is in $[H_j]$.

There must be a slot m from which x was removed. By Lemma 1.4, all x_l are stored in slots before slot m.

Combining the above two sentences, we have: at the time x was removed from the slot m, there is a chain with length $\ge K$ between slot 1 and $m - 1$, which contradicts Lemma 3.2. ■

THEOREM 3. *If H is a complete history accepted by the semiqueue implementation, then H is linearizable.*

Proof. Similar to that for the unbounded FIFO queue using Lemmas 1.1–1.4 and 3.1–3.3. ■

8. STUTTERING QUEUES

8.1. Specification

A *Stuttering$_j$–Queue* object (Fig. 8) is like a FIFO queue except that the first item in the queue may be returned up to j times. As for the semiqueue, the motivation for allowing stuttering is to give quicker response to concurrent dequeuers. Relaxing the strict FIFO constraint of a normal queue means letting up to j dequeueing processes get the same first element. In the case that j is always equal to 1, i.e., that there is always only one active dequeueing process, then a stuttering queue behaves like a FIFO queue.

```
StutQ: trait
  includes FifoQ
  StQ record of [items: Q, count: Int]

void Enq(queue q, elt e) {
   modifies q
   ensures q'.items = ins(q.items, e)
   }

elt Deq(queue q) {
   requires ¬ isEmp(q.items)
   modifies q
   ensures
      q.count < j ⇒ (result = first(q.items) ∧
                  ((q'.count = q.count + 1 ∧ q'.items = q.items) ∨
                  (q'.count = 0 ∧ q'.items= rest(q.items) ) ) )
   }
```

FIG. 8. *Stuttering$_j$—Queue.*

8.2. Implementation

In our implementation we do not use the atomic instructions SWAP and ADD, but instead we use FETCH_AND_ADD, STORE, and SUB:

```
void SUB(int *p) {
  modifies *p
  ensures (*p)' = (*p) − 1
  }
```

SUB decreases by one the value of the integer pointed to by p.

Again, only the differences to the *Deq* operation are of interest:

```
elt Deq(queue q) {
    int i, range, num;
    elt value;
    slot current;

    while (true) {
        current = q.head;
        range = READ(&q.back);
        for (i = 1; i <= range; i ++) {
        num = FETCH_AND_ADD(&q,&(current->hold));
           /* get the number of processes currently
                        looking at the same slot */
            if (num < J) {
           /* if it is not greater than J */
                value = current->item;
           /* not atomic because want
                                          to allow */
           /* allow other process to get
                                        same value */
                if (value != NULL) {
           /* successful dequeue */
                    current->item = NULL;
           /* set this location
                                    to NULL value */
                    SUB(&(current->hold));
                    return(value);
                    }
                }
              SUB(&(current->hold));
           /* There have been J processes
                              looking at this slot */
           /* so this process should not
                            try to dequeue it again */
              current = current->next;
           /* try next slot */
              }
        }
    }
```

As with the implementation of the FIFO queue, we use *back* to limit the search range of a *Deq* operation. We use *hold* to remember how many processes are currently "looking at" the item stored in a slot. Whenever a process wants to search for an item from a slot, it first checks and increases this value. J determines the maximum number of processes that are allowed to look at the same slot concurrently and hence, the maximum times an item could be returned. In contrast to the FIFO queue implementation, here when a *Deq* operation gets a non-NULL value from a slot it does not immediately swap in the NULL element, thus giving other dequeueing processes the chance to get the same item from the slot. However, the maximum number of processes that can get an item from one slot is limited to be less than J. By carefully ordering the updates to *hold* and *item*, we ensure that before *hold* of a slot is decreased, the *item* has been initialized to *NULL*, prohibiting more processes from getting this item from this slot.

8.3. Proof of Correctness for the Stuttering Queue

Lemmas 1.2, 1.3, and 1.4 hold for the *Stuttering$_j$-Queue*.

LEMMA 4.1. *No item can be dequeued more than J times in the Stuttering$_j$-Queue.*

Proof. By observing the following two facts: (1) We allow at most J processes to access a slot concurrently (determined by the value of *hold* of the slot); (2) If exactly one process has gotten an item from the mth slot and decreases *hold* value in that slot, then slot m will be set to NULL. So several processes can get an item from slot m only if they access slot m before any one of them has decreased the *hold* value of that slot; however, only J processes are allowed to do so. ■

LEMMA 4.2. *When a process P gets an item x from the mth slot, x must be the minimal item in the set $S = \{y \mid y$ is stored in some slot between the first and mth slots and there are fewer than J processes that have also dequeued $y\}$.*

Proof. Since each time P must first get the value of *back* in determining its *range* and then start its search from the first slot, for any $y \in S$, it must be true that $label(enq(y)) < label(deq(x))$. If there is an item $y \in S$ such that $label(eok(y)) < label(enq(x))$, then $label(eok(y)) < label(deq(x))$, and also y was stored in a slot before the slot in which x is stored (by Lemma 1.4). Thus, P could get a non-NULL item y before reaching x, a contradiction. ■

LEMMA 4.3. *If H is a history accepted by the Stuttering$_j$-Queue and x is an item such that $H_i = deq(x)$ and $H_j = dok(x)$, then $x \in enqueue([H_{j-1}])$ and x is the minimal item in $left([H_{j-1}])$*

Proof. If not, there must a $y \in left([H_{j-1}])$ such that $label(eok(y)) < label(enq(x))$ and it must be true that $y \in S$ (defined as in Lemma 4.2) since $left(H) \subseteq S$. By Lemma 4.2, this is impossible. ■

THEOREM 4. *If H is a complete history accepted by the stuttering queue implementation, then H is linearizable.*

Proof. Similar to the proof for the unbounded FIFO queue using Lemmas 1.2–1.4 and 4.1–4.3. ■

9. CURRENT AND FUTURE WORK

Building the simulator and collecting a library of verified concurrent objects are first steps toward understanding linearizability as a correctness condition from both practical and theoretical viewpoints. We are interested in pursuing two directions of further work: (1) investigating more structured proof methods for showing the correctness of implementations, and (2) using the simulator to help fine-tune correct implementations.

We observe that the critical insight needed in the following proofs is in defining the partial order, $<_r$. One way to do a more syntax-directed verification of our code, as suggested by Herlihy and Wing [15], is to encode this partial order as auxiliary data and reason about this hidden state information. Such a proof method yields cumbersome proofs and tends to obscure the critical insight. We are currently exploring simpler and more elegant proof methods. For example, we are investigating the relevance of Brookes's work on syntax-directed proof methods for reasoning about concurrent programs [1, 2]. We hope to be able to develop an axiomatic proof method that lets one reason about the underlying semantic structure of the model of computation. The major difficulty seems to be the need to handle the potentially complex interactions of concurrent objects without recourse to brute-force analysis based on interleaving.

We would like to extend our simulation environment to serve as a basis for measuring the performance of implementations on multi-processor architectures. This basis would be useful for producing statistics to compare the relative efficiency of different implementations and to compare a concurrent object under different loads and test conditions (e.g., many dequeueing processes). We have taken preliminary performance statistics on all objects in our library on both single and multiprocessor architectures. They indicate that we need much better instrumentation of the lower-level operating system primitives, e.g., those provided by C-Threads, before we can draw definitive conclusions on relative performance of our implementations.

APPENDIX: THE SEARCH ALGORITHM

```
/* search for the the next possible operation in the history H */

event *search(event *h) {
  elt_stack current;  /* Used to locate the current section and the selected operation. */
  elt_stack *tmp;     /* Auxiliary pointer */
  event *head;        /* Head pointer of the remaining subhistory */
  bool found;

  init_stack(&stack);              /* Initialize stack. */
  head = h;
  while (head->next != NULL) {     /* The remaining subhistory is not empty. */
    current.pi = head;             /* Let the first section of the remaining */
    current.pr = head->match;      /* subhistory be the current section. */
    current.inv = current.pi;      /* Try the first operation in the current section. */
    while (true) {                 /* Keep trying until success. */
      op_copy(current.inv, &current); /* Store the selected operation in current. */
      if (operation(current.op, current.item, current.result, 1)) {
            /* This operation is allowed by the sequential object. */
       push(&stack, &current);         /* Push current into stack. */
       if (lift(current.inv, current.resp))  { /* Lift the selected operation from H. */
         head = current.pi->next;              /* Update head if it's the */
         }                                     /* first operation of the history. */
         break;
       }
      else    /* This operation is not allowed. */
        do {  /* Looking for another operation within the current section. */
          current.inv = current.inv->next;
          found = true;
          if ((current.inv->op == 'O') || (current.inv == current.pr)) {
           /* Every operation in the current section has been tried without success. */
            found = false;
            if (isempty(&stack)) {           /* If no previous section */
              return(NULL);                  /* return NULL value. */
              }
            tmp = top(&stack);               /* bBacktrack to the previous section. */
            pop(&stack);
            if (operation(tmp->op, tmp->item, tmp->result, 0)) {
                /* undo the previous selected operation */
              unlift(tmp->inv, tmp->resp); /* Put it back in history. */
              current.pi = tmp->pi;        /* The previous section */
              current.pr = tmp->pr;        /* becomes the current section. */
              head = tmp->pi;
              current.inv = tmp->inv;
              }
          }
       } while (!found);
    }
 }
 current.inv = head;
 push(&stack, &current);            /* Push an extra event as sentinel. */
 return(linearization(&stack));     /* Return linearized history. */
}
```

ACKNOWLEDGMENTS

The authors thank Maurice Herlihy for earlier collaborative work on linearizability and Francesmary Modugno for suggesting a polynomial-time analysis algorithm for the FIFO queue. We also thank the anonymous referees for their helpful and insightful comments.

This research was supported in part by the National Science Foundation under Grant CCR-8906483 under the special NSF/DARPA joint initiative on Parallel Computing Theory and in part by the Office of Naval Research under Grant N00014-88-K-0699.

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 of the U.S. Government.

REFERENCES

1. Brookes, S. D. An axiomatic treatment of a parallel language. In *Proceedings of Conference on Logics of Programs*. Brooklyn, NY, 1985. Springer Lecture Notes in Computer Science, Vol. 193, Springer-Verlag, Berlin/New York, 1985.
2. Brookes, S. D. A semantically-based proof system for partial correctness and deadlock in CSP. In *Proceedings of Symposium on Logic in Computer Science*. Cambridge, MA, 1986. IEEE Computer Society Press, New York, 1986.
3. Burns, J. E., and Peterson, G. L. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, 1987, pp. 222–231.
4. Chor, B., Israeli, A., and Li, M. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, 1987, pp. 86–97.
5. Cooper, Eric C. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, 1988.
6. Dijkstra, E. W. *Notes on Structured Programming*. Academic Press, New York, 1972, pp. 1–82.
7. Ellis, C. S. Concurrent search and insertion in 2–3 trees. *Acta Inform.* **14** (1980).
8. Gottlieb, A., Lubachevsky, B. D. and Rudolph, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Programming Lang. Systems* **5**, 2 (Apr. 1983), 164–189.
9. Schneider, F. B., and Andrews, G. R. Concepts and notations for concurrent programming. *Comput. Surveys* **15**, 1 (Mar. 1983), 1–43.
10. Guttag, J. V., and Horning, J. J. Introduction to LCL, a Larch/C interface language. Technical Report 74, DEC/Systems Research Center, Aug. 1991.
11. Guttag, J. V., Horning, J. J., and Wing, J. M. The Larch family of specification languages. *IEEE Software* **2**, 5 (Sept. 1985), 24–36.
12. Herlihy, M. P. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. Aug. 1988.
13. Herlihy, M. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 1990, pp. 197–206.
14. Herlihy, M. P., and Wing, J. M. Implementing queues without mutual exclusion. Some queue examples.
15. Herlihy, M. P., and Wing, J. M. Axioms for concurrent objects. In *Fourteenth ACM Symposium on Principles of Programming Languages*, Jan. 1987, pp. 13–26.
16. Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Programming Lang. Systems* **12**, 3 (July 1990), 463–492.
17. Herlihy, M. P., and Wing, J. M. Specifying graceful degradation. *IEEE Trans. Parallel Distrib. Comput.* (Jan. 1991), 93–104.
18. Lamport, L. Concurrent reading and writing. *Comm. ACM* **20**, 11 (Nov. 1977), 806–811.
19. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* **C-28**, 9 (Sept 1979), 690.
20. Lanin, V., and Shasha, D. Concurrent set manipulation without locking. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*. Mar. 1988, pp. 211–220.
21. Lehman, P. L., and Bing Yao, S. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems* **6**, 4 (1981), 650–670.
22. Mellor-Crummey, J. M. Concurrent queues: Practical *fetch_and_*ϕ algorithms. Technical Report TR 229, Dept. of Computer Science, University of Rochester, Nov. 1987.
23. Misra, J. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Programming Lang. Systems* **8**, 1 (Jan. 1986), 142–153.
24. Papadimitriou, C. H. The serializability of concurrent database updates. *J. Assoc. Comput. Math.* **26**, 4 (Oct. 1979), 631–653.
25. Weihl, W. E., and Wang, P. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the 1990 IEEE Symposium on Parallel and Distributed Computing*. Dallas, TX, Dec. 1990, pp. 650–655.

Received January 8, 1991; revised August 13, 1991; accepted November 22, 1991

JEANNETTE M. WING is an Associate Professor of Computer Science at Carnegie Mellon University. Her research interests include formal specifications, programming languages, and concurrent and distributed systems. She heads the Venari Project, whose goal is to provide support for distributed object management. She was actively involved in the design of the Avalon transaction-based programming language and the Miro visual specification language. She continues to contribute to the design of the Larch family of specification languages. Wing received her S.B., S.M., and Ph.D. degrees in computer science from the Massachusetts Institute of Technology. She is a member of the IEEE and ACM.

CHUN GONG received a B.S. in computer science from Peking University, People's Republic of China, in 1982, and an M.S. in computer science from the Institute of Software, Academia Sinica, People's Republic of China, in 1985. From 1985 to 1988, he was research assistant at the Institute of Software, Academia Sinica. From 1988 to 1990, he was visiting the School of Computer Science, Carnegie Mellon University. Since January 1991, he has been at the University of Pittsburgh studying for his Ph.D. His research interests include semantics of programming languages, parallel and distributed computing systems, and parallel algorithms.